



US009460290B2

(12) **United States Patent**  
**Glew et al.**

(10) **Patent No.:** **US 9,460,290 B2**  
(45) **Date of Patent:** **\*Oct. 4, 2016**

(54) **CONDITIONAL SECURITY RESPONSE  
USING TAINT VECTOR MONITORING**

(75) Inventors: **Andrew F. Glew**, Portland, OR (US);  
**Daniel A. Gerrity**, Seattle, WA (US);  
**Clarence T. Tegreene**, Mercer Island,  
WA (US)

(73) Assignee: **Elwha LLC**, Bellevue, WA (US)

(\*) Notice: Subject to any disclaimer, the term of this  
patent is extended or adjusted under 35  
U.S.C. 154(b) by 325 days.  
  
This patent is subject to a terminal dis-  
claimer.

(21) Appl. No.: **13/317,980**

(22) Filed: **Oct. 31, 2011**  
(Under 37 CFR 1.47)

(65) **Prior Publication Data**  
US 2013/0024939 A1 Jan. 24, 2013

**Related U.S. Application Data**

(63) Continuation-in-part of application No. 13/136,024,  
filed on Jul. 19, 2011, and a continuation-in-part of  
application No. 13/136,400, filed on Jul. 29, 2011,  
now Pat. No. 8,930,714, and a continuation-in-part of

(Continued)

(51) **Int. Cl.**  
**G06F 11/00** (2006.01)  
**G06F 21/57** (2013.01)  
**H04L 29/06** (2006.01)

(52) **U.S. Cl.**  
CPC ..... **G06F 21/577** (2013.01); **H04L 63/1408**  
(2013.01); **H04L 63/1416** (2013.01)

(58) **Field of Classification Search**

CPC ..... H04L 63/1416; H04L 63/1408  
USPC ..... 726/22-25  
See application file for complete search history.

(56) **References Cited**

**U.S. PATENT DOCUMENTS**

4,087,856 A	5/1978	Attanasio
4,525,599 A	6/1985	Curran et al.
5,437,032 A	7/1995	Wolf et al.
5,623,637 A	4/1997	Jones et al.
6,006,328 A	12/1999	Drake
6,057,598 A	5/2000	Payne et al.
6,145,064 A	11/2000	Long et al.
6,154,741 A	11/2000	Feldman
6,199,055 B1	3/2001	Kara et al.
6,205,544 B1	3/2001	Mills et al.

(Continued)

**FOREIGN PATENT DOCUMENTS**

WO WO 0199075 A2 \* 12/2001

**OTHER PUBLICATIONS**

Stone, Harold S.; "A Logic-in-Memory Computer"; IEEE Transac-  
tions on Computers; bearing a date of Jan. 1970; pp. 73-78.

(Continued)

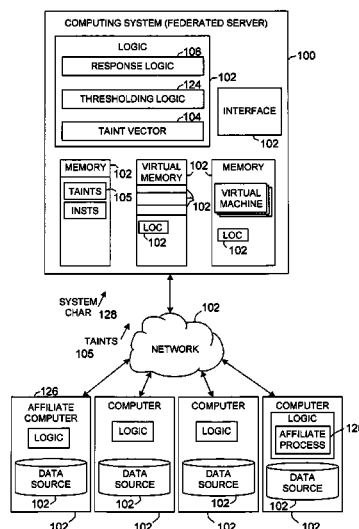
*Primary Examiner* — Chris Parry

*Assistant Examiner* — Weiwei Stiltner

(57) **ABSTRACT**

An embodiment or embodiments of a computing system can conditionally trap based on a taint vector. A computing system can comprise at least one taint vector operable to list at least one of a plurality of taints indicative of potential security risk originating from at least one of a plurality of resources, and response logic operable to monitor the at least one taint vector and respond to a predetermined taint condition.

**47 Claims, 21 Drawing Sheets**



**Related U.S. Application Data**

application No. 13/136,401, filed on Jul. 29, 2011, now Pat. No. 8,943,313, and a continuation-in-part of application No. 13/136,666, filed on Aug. 4, 2011, and a continuation-in-part of application No. 13/136,670, filed on Aug. 4, 2011, and a continuation-in-part of application No. 13/199,368, filed on Aug. 26, 2011, and a continuation-in-part of application No. 13/200,547, filed on Sep. 24, 2011, now Pat. No. 8,955,111, and a continuation-in-part of application No. 13/200,557, filed on Sep. 24, 2011, now Pat. No. 9,170,843, and a continuation-in-part of application No. 13/200,550, filed on Sep. 24, 2011, and a continuation-in-part of application No. 13/200,556, filed on Sep. 24, 2011, and a continuation-in-part of application No. 13/317,834, filed on Oct. 28, 2011, now Pat. No. 9,098,608, and a continuation-in-part of application No. 13/317,826, filed on Oct. 28, 2011, now Pat. No. 8,813,085, and a continuation-in-part of application No. 13/317,825, filed on Oct. 28, 2011.

(56)

**References Cited****U.S. PATENT DOCUMENTS**

6,584,488 B1	6/2003	Brenner et al.	8,688,583 B2	4/2014	Boccon-Gibod et al.
6,654,745 B2	11/2003	Feldman	8,694,947 B1	4/2014	Venkataramani et al.
6,996,547 B1	2/2006	Tugenberg et al.	8,732,431 B2	5/2014	Culley et al.
7,035,277 B1	4/2006	Batcher	2002/0040420 A1	4/2002	Yamauchi et al.
7,054,190 B2	5/2006	Hanyu et al.	2002/0075844 A1	6/2002	Hagen
7,069,447 B1	6/2006	Corder	2002/0141577 A1	10/2002	Ripley et al.
7,093,250 B1	8/2006	Rector	2002/0142833 A1	10/2002	Tsuchida et al.
7,107,176 B2	9/2006	Henry et al.	2002/0152212 A1	10/2002	Feldman
7,124,170 B1	10/2006	Sibert	2002/0156939 A1	10/2002	Armstrong et al.
7,165,150 B2	1/2007	Alverson et al.	2002/0166058 A1	11/2002	Fueki
7,191,185 B2	3/2007	Dweck et al.	2003/0046238 A1	3/2003	Nonaka et al.
7,221,600 B2	5/2007	Hara et al.	2003/0084308 A1	5/2003	Van Rijnsouw
7,284,000 B2	10/2007	Kuehr-McLaren et al.	2003/0088759 A1	5/2003	Wilkerson
7,379,999 B1 *	5/2008	Zhou et al. .... 709/224	2003/0149869 A1	8/2003	Gleichauf
7,395,414 B2	7/2008	Le et al.	2003/0159070 A1	8/2003	Mayer et al.
7,502,946 B2	3/2009	Perkins et al.	2003/0182436 A1	9/2003	Henry
7,533,242 B1	5/2009	Moll et al.	2003/0187974 A1	10/2003	Burbeck et al.
7,533,273 B2	5/2009	Patariu et al.	2003/0188132 A1	10/2003	Keltcher et al.
7,549,054 B2	6/2009	Brodie et al.	2004/0054925 A1	3/2004	Etheridge et al.
7,594,111 B2	9/2009	Kiriansky et al.	2004/0088691 A1	5/2004	Hammes et al.
7,620,941 B1	11/2009	Leventhal	2004/0116183 A1	6/2004	Prindle
7,644,162 B1	1/2010	Zhu et al.	2004/0117639 A1	6/2004	Mowery
7,676,578 B1	3/2010	Zhu et al.	2004/0117790 A1	6/2004	Rhine
7,708,195 B2	5/2010	Yoshida et al.	2004/0153318 A1 *	8/2004	Chamberlain ..... 704/222
7,757,282 B2	7/2010	Pandit et al.	2004/0199763 A1	10/2004	Freund
7,844,733 B2	11/2010	Betts et al.	2004/0236958 A1	11/2004	Teicher et al.
7,861,305 B2	12/2010	McIntosh et al.	2005/0004924 A1	1/2005	Baldwin
7,870,610 B1	1/2011	Mitchell et al.	2005/0060710 A1	3/2005	Kush
7,873,998 B1	1/2011	Wilkinson et al.	2005/0081207 A1	4/2005	Hoflehner et al.
7,877,585 B1	1/2011	Coon et al.	2005/0125613 A1	6/2005	Kim et al.
7,953,986 B2	5/2011	Lee	2005/0125793 A1	6/2005	Aguilar, Jr. et al.
7,958,370 B2	6/2011	Hirai et al.	2005/0138074 A1	6/2005	O'Connor et al.
7,958,558 B1	6/2011	Leake et al.	2005/0160428 A1	7/2005	Ayachitula et al.
8,022,724 B1	9/2011	Jenkins, IV	2005/0171903 A1	8/2005	Yacobi et al.
8,099,574 B2	1/2012	Savagaonkar et al.	2005/0177596 A1	8/2005	Wu et al.
8,136,158 B1	3/2012	Sehr et al.	2005/0195975 A1	9/2005	Kawakita
8,146,106 B2	3/2012	Kim et al.	2005/0213751 A1	9/2005	Apostolopoulos et al.
8,281,388 B1	10/2012	Sobel et al.	2005/0232415 A1	10/2005	Little et al.
8,286,250 B1	10/2012	Le et al.	2005/0268075 A1	12/2005	Caprioli et al.
8,312,509 B2	11/2012	Zimmer et al.	2006/0005082 A1	1/2006	Fossum et al.
8,381,192 B1 *	2/2013	Drewry et al. .... 717/128	2006/0021054 A1	1/2006	Costa et al.
8,397,238 B2	3/2013	Venkumahanti et al.	2006/0025952 A1	2/2006	Buhr
8,473,754 B2	6/2013	Jones et al.	2006/0161715 A1	7/2006	Hamaguchi
8,510,827 B1 *	8/2013	Leake et al. .... 726/22	2006/0161978 A1	7/2006	Abadi et al.
8,516,583 B2	8/2013	Thomas et al.	2006/0184767 A1	8/2006	Le et al.
8,555,390 B2	10/2013	Thiebaud de la Crouee et al.	2006/0289659 A1	12/2006	Mizushima
8,621,144 B2	12/2013	Eschmann et al.	2007/0022287 A1	1/2007	Beck et al.
8,675,868 B1	3/2014	Yearsley et al.	2007/0050477 A1	3/2007	Isaacs
8,683,581 B2	3/2014	Lefloch	2007/0079304 A1	4/2007	Zheng et al.
			2007/0101433 A1	5/2007	Louch et al.
			2007/0240215 A1	10/2007	Flores et al.
			2007/0250837 A1	10/2007	Herington et al.
			2008/0005586 A1	1/2008	Munguia
			2008/0028467 A1	1/2008	Kommareddy et al.
			2008/0046997 A1	2/2008	Wang
			2008/0052539 A1	2/2008	MacMillan et al.
			2008/0052541 A1	2/2008	Ginter et al.
			2008/0072075 A1	3/2008	Kohiyama et al.
			2008/0104004 A1 *	5/2008	Brave et al. .... 706/45
			2008/0114990 A1	5/2008	Hilbert et al.
			2008/0127335 A1	5/2008	Khan et al.
			2008/0168279 A1	7/2008	Kanai
			2008/0184016 A1	7/2008	Erlingsson et al.
			2008/0222397 A1	9/2008	Wilkerson et al.
			2008/0222532 A1	9/2008	Mester et al.
			2008/0250216 A1	10/2008	Kershaw et al.
			2008/0256346 A1	10/2008	Lee et al.
			2008/0263663 A1	10/2008	Ide et al.
			2008/0276317 A1	11/2008	Chandola et al.
			2008/0279371 A1	11/2008	Lee et al.
			2008/0288785 A1	11/2008	Rao et al.
			2008/0301467 A1	12/2008	Saito
			2008/0313482 A1	12/2008	Karlalalem et al.
			2008/0320565 A1	12/2008	Buch et al.
			2009/0006755 A1	1/2009	Illikkal et al.
			2009/0007125 A1	1/2009	Barsness et al.
			2009/0028135 A1	1/2009	Mantripragada et al.
			2009/0038014 A1	2/2009	Force et al.
			2009/0063824 A1	3/2009	Leaback et al.

(56)

**References Cited****U.S. PATENT DOCUMENTS**

2009/0070338	A1	3/2009	Spitzig et al.	
2009/0083520	A1	3/2009	Kanemura	
2009/0106563	A1	4/2009	Cherpantier	
2009/0144557	A1	6/2009	Sutton	
2009/0172056	A1	7/2009	Pradhan et al.	
2009/0172686	A1	7/2009	Chen et al.	
2009/0183263	A1	7/2009	McMichael et al.	
2009/0187743	A1	7/2009	Greenhalgh	
2009/0187771	A1	7/2009	McLellan, Jr.	
2009/0210881	A1	8/2009	Duller	
2009/0222910	A1	9/2009	Le Bihan et al.	
2009/0265712	A1	10/2009	Herington	
2009/0282474	A1	11/2009	Chen et al.	
2009/0288090	A1	11/2009	Ujibashi et al.	
2009/0320129	A1	12/2009	Pan et al.	
2009/0320136	A1	12/2009	Lambert et al.	
2009/0327718	A1	12/2009	Hirai	
2010/0017638	A1	1/2010	Ghose	
2010/0042824	A1	2/2010	Lee et al.	
2010/0131957	A1	5/2010	Kami	
2010/0165991	A1*	7/2010	Veal et al.	370/392
2010/0191349	A1	7/2010	Munaga	
2010/0269168	A1*	10/2010	Hegli et al.	726/11
2010/0281273	A1	11/2010	Lee et al.	
2010/0299305	A1*	11/2010	Laxman et al.	707/609
2010/0318998	A1	12/2010	Golla	
2011/0029140	A1*	2/2011	Jordan et al.	700/286
2011/0066896	A1	3/2011	Ebina et al.	
2011/0072292	A1	3/2011	Khawand et al.	
2011/0129024	A1*	6/2011	Karthik et al.	375/260
2011/0131402	A1	6/2011	Mittal	
2011/0131550	A1	6/2011	Burckhardt et al.	
2011/0131658	A1*	6/2011	Bahl	726/25
2011/0138124	A1	6/2011	Hill et al.	
2011/0138473	A1	6/2011	Yee et al.	
2011/0179176	A1*	7/2011	Ravichandran et al.	709/226
2011/0202927	A1	8/2011	Miloushev et al.	
2011/0289586	A1	11/2011	Kc et al.	
2011/0296440	A1	12/2011	Laurich et al.	
2011/0307382	A1*	12/2011	Siegel et al.	705/44
2011/0314254	A1*	12/2011	Smriti et al.	712/7
2011/0320788	A1	12/2011	Assarpour	
2011/0320793	A1	12/2011	Bell, Jr. et al.	
2012/0079492	A1	3/2012	Chambliss et al.	
2012/0102489	A1	4/2012	Staiman et al.	
2012/0110292	A1	5/2012	Martini	
2012/0159183	A1	6/2012	Adams et al.	
2012/0179952	A1	7/2012	Tuyls et al.	
2012/0185863	A1	7/2012	Krstic et al.	
2012/0210325	A1	8/2012	de Lind van Wijngaarden et al.	
2012/0221591	A1*	8/2012	Yerneni et al.	707/769
2012/0233698	A1	9/2012	Watters et al.	
2012/0255018	A1	10/2012	Sallam	
2012/0255021	A1*	10/2012	Sallam	726/25
2012/0265975	A1	10/2012	Kimelman	
2012/0266243	A1	10/2012	Turkulainen	
2012/0278903	A1	11/2012	Pugh	
2013/0024867	A1	1/2013	Glew et al.	
2013/0081039	A1	3/2013	Glew et al.	
2013/0086687	A1	4/2013	Chess et al.	
2013/0111489	A1	5/2013	Glew et al.	
2013/0111491	A1	5/2013	Glew et al.	
2013/0246605	A1*	9/2013	Mahadik et al.	709/224
2013/0305243	A1	11/2013	Hiki	
2014/0245449	A1	8/2014	Powell et al.	

**OTHER PUBLICATIONS**

Bird et al.; "Lighthouse: Hardware Support for Enforcing Information Flow Control on ManyCore Systems"; bearing a date of 2008 (as provided by examiner); 10 pages.

Clause et al.; "Dytan: A Generic Dynamic Taint Analysis Framework"; ISSSTA '07; bearing a date of Jul. 9-12, 2007; pp. 196-206; ACM.

Ho et al.; "Practical Taint-Based Protection using Demand Emulation"; EuroSys '06; bearing a date of Apr. 18-21, 2006; 13 pages; ACM.

Newsome et al.; "Dynamic Taint Analysis for Automation Detection, Analysis, and Signature Generation of Exploits on Commodity Software"; Carnegie Mellon University Research Showcase; bearing a date of Jan. 1, 2005; 18 pages.

Schwartz et al.; "All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but might have been afraid to ask)"; 2010 IEEE Symposium on Security and Privacy; bearing a date of 2010; pp. 317-331; IEEE.

Song et al.; "BitBlaze: A New Approach to Computer Security via Binary Analysis"; ICISS 2008; bearing a date of 2008; pp. 1-25; Springer-Verlag Berlin Heidelberg.

Yin et al.; "Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis"; CCS '07; bearing a date of Oct. 29-Nov. 2, 2007; pp. 116-127; ACM.

Zeldovich et al.; "Hardware Enforcement of Application Security Policies Using Tagged Memory"; bearing a date of 2008 (as provided by examiner), printed Feb. 7, 2013; pp. 1-16; located at: [http://static.usenix.org/events/osdi08/tech/full\\_papers/zeldovich/zeldovich.html](http://static.usenix.org/events/osdi08/tech/full_papers/zeldovich/zeldovich.html).

Zhang et al.; "Neon: System Support for Derived Data Management"; VEE '10; bearing a date of Mar. 17-19, 2010; 12 pages; ACM.

Adya et al.; "FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment"; 5<sup>th</sup> Symposium on Operating Systems Design and Implementation; bearing a date of Dec. 2002; pp. 1-14; Boston, MA.

Kolbitsch et al.; "Extending Mondrian Memory Protection"; Information Systems and Technology Panel (IST) Symposium, Tallinn, Estonia; bearing a date of Nov. 22-23, 2010; pp. 10-1 through 10-18 and 1 Report Documentation Page; NATO-OTAN.

Abadi et al.; "Control-Flow Integrity: Principles, Implementations, and Applications"; bearing a date of Nov. 1, 2004; pp. 1-33.

Kiriansky et al.; "Secure Execution via Program Shepherding"; Proceedings of the 11<sup>th</sup> USENIX Security Symposium; bearing a date of Aug. 2002, created on Dec. 9, 2013; pp. 1-16; San Francisco, CA.

"Instruction Set"; Wikipedia; bearing a date of May 15, 2010; pp. 1-10; located at: [https://en.wikipedia.org/wiki/Instruction\\_set](https://en.wikipedia.org/wiki/Instruction_set).

"Opcode"; Wikipedia; bearing a date of Oct. 26, 2014; pp. 1-2; located at: <http://en.wikipedia.org/w/index.php?title=Opcode&oldid=175723708>.

Chen et al.; "Log-Based Architectures for General-Purpose Monitoring of Deployed Code"; bearing a date of Oct. 21, 2006; 3 pages; ACM.

Liu et al.; "Adaptive Entitlement Control of Resource Containers on Shared Servers"; bearing a date of 2005; pp. 163-176; IEEE.

Liu et al.; "Optimal Multivariate Control for Differentiated Services on a Shared Hosting Platform"; Proceedings of the 46<sup>th</sup> IEEE Conference on Decision and Control; bearing a date of Dec. 12-14, 2007; pp. 3792-3799; IEEE.

Wang et al.; "AppRAISE: Application-Level Performance Management in Virtualized Server Environments"; bearing a date of Dec. 2009; pp. 240-254; vol. 6, No. 4; IEEE.

Realtime Privacy Monitoring on Smartphones; located at <http://appanalysis.org>; (retrieved on Oct. 8, 2015) created on Dec. 2, 2015; pp. 1-2.

Enck et al.; "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones"; Proceedings of the 9<sup>th</sup> USENIX Symposium on Operating Systems Design and Implementation (OSDI), Oct. 2010. Vancouver, BC.; created on Dec. 2, 2015; pp. 1-15.

\* cited by examiner

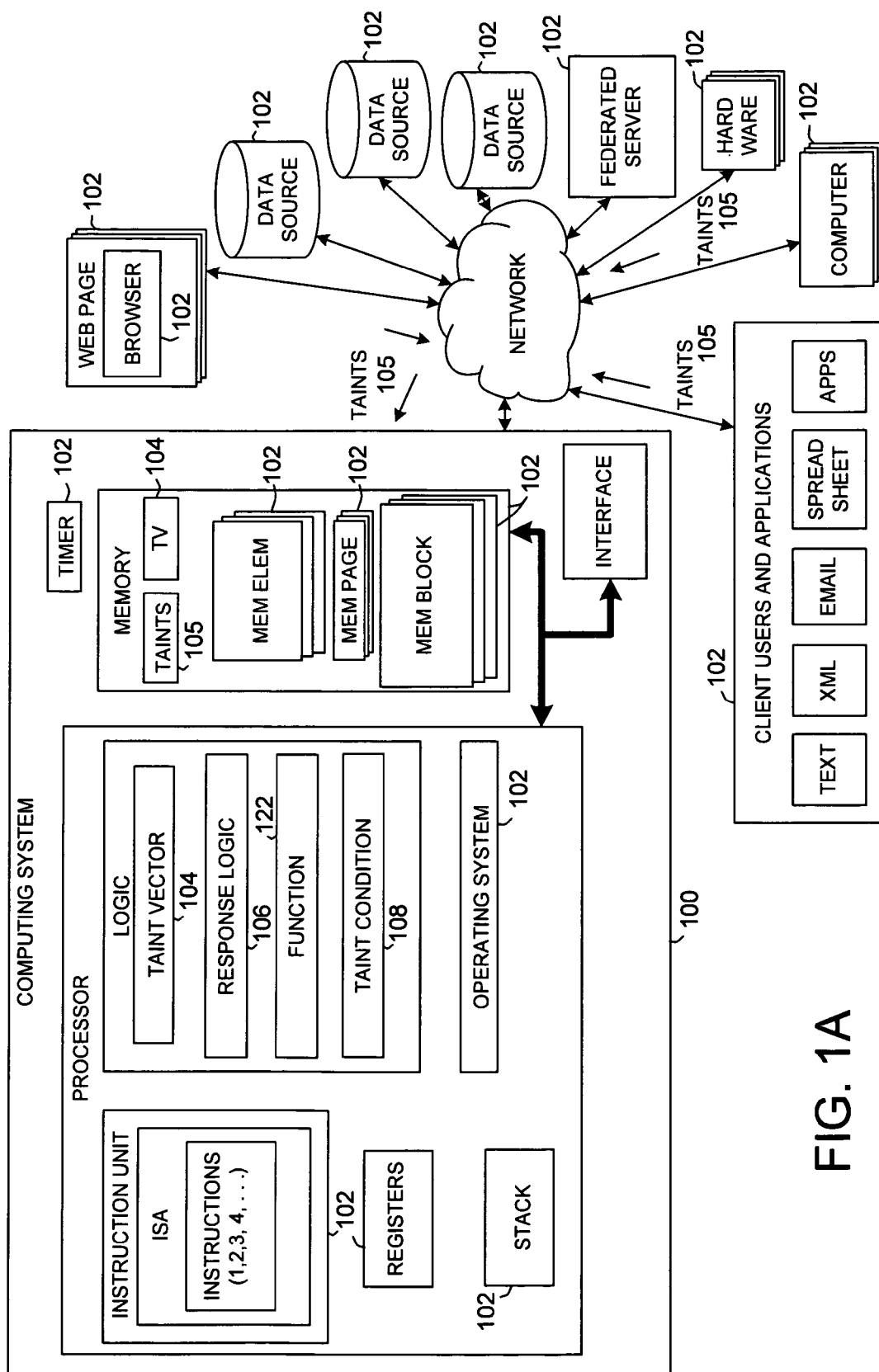


FIG. 1A

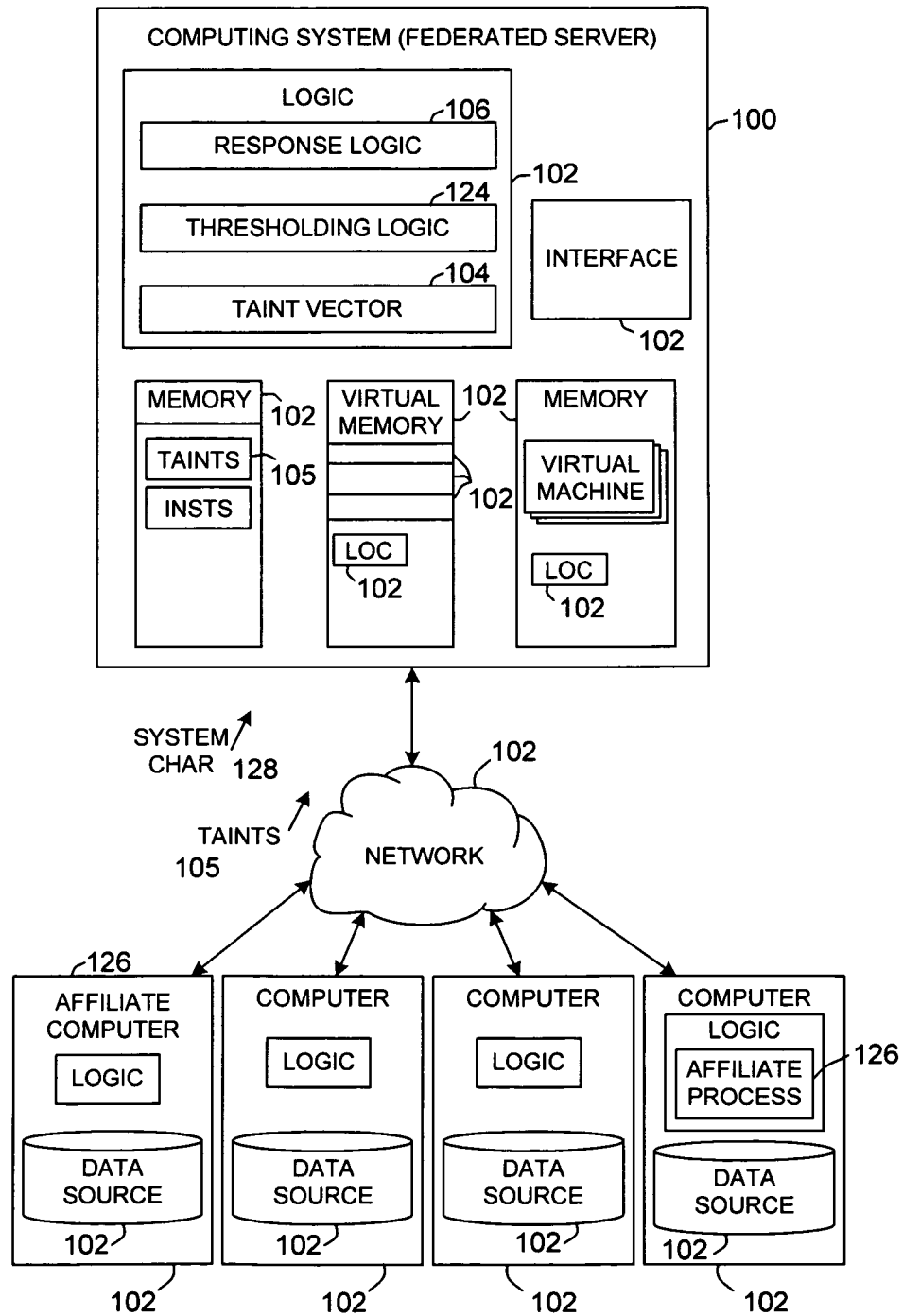


FIG. 1B

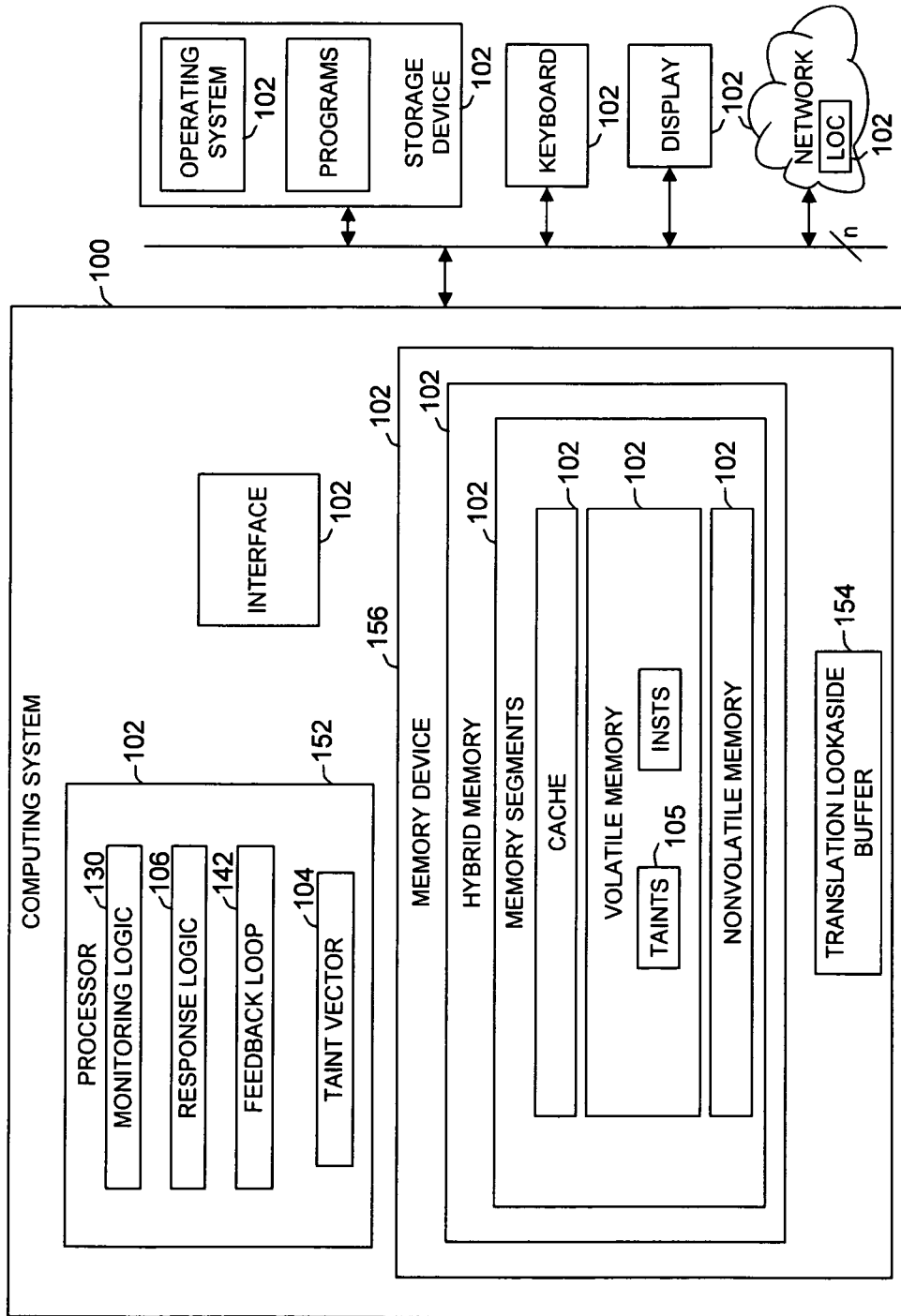


FIG. 1C

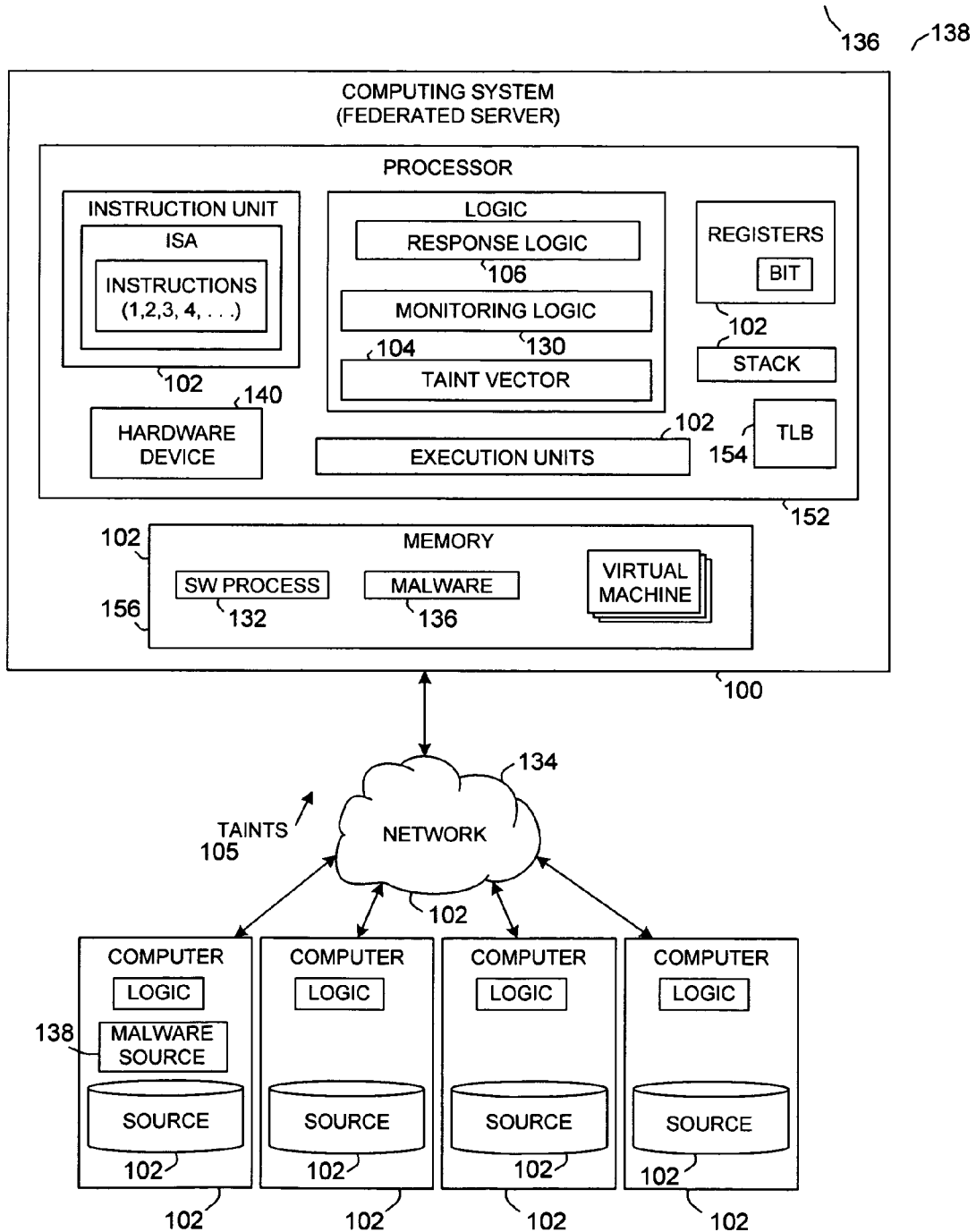


FIG. 1D

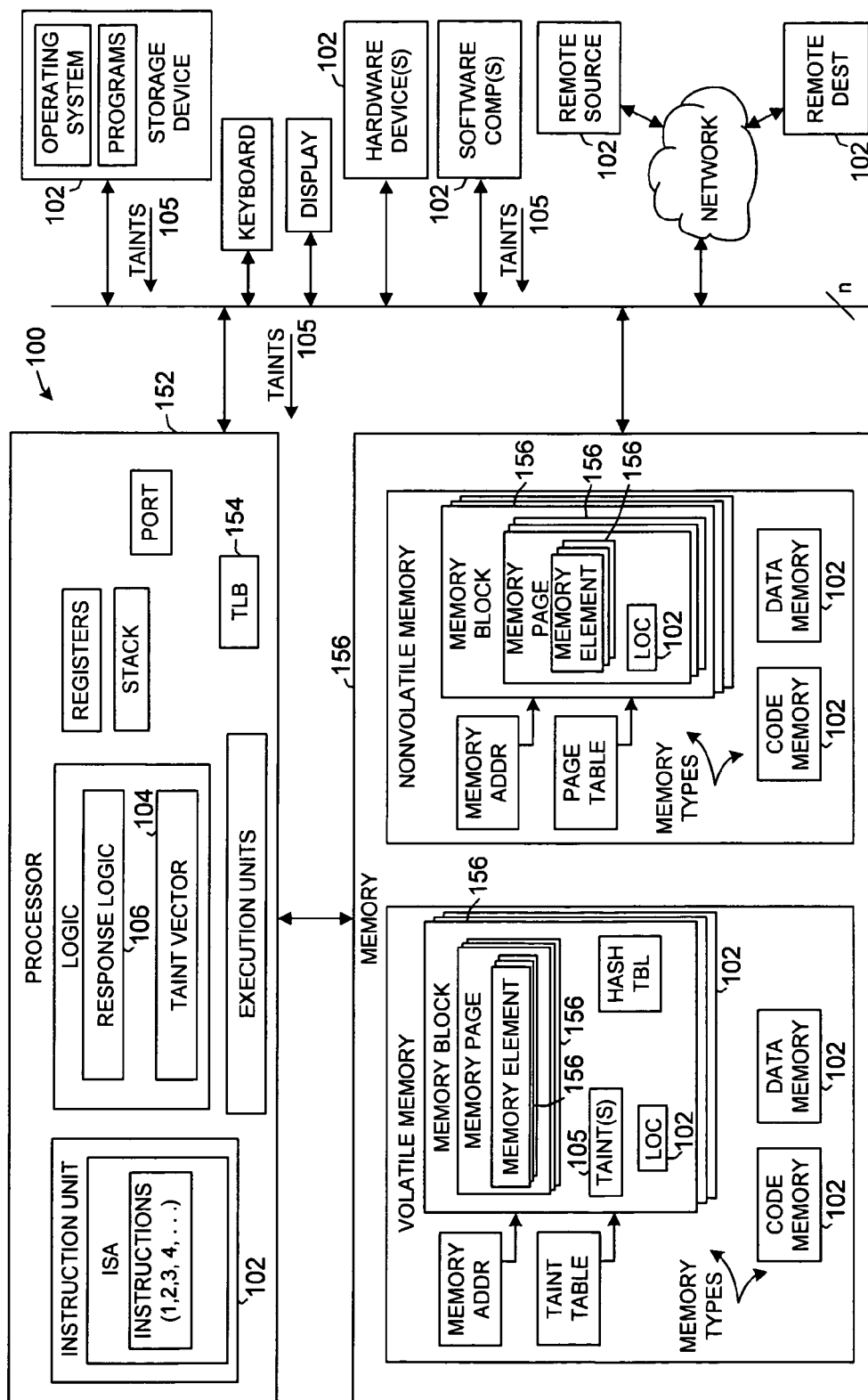


FIG. 1E



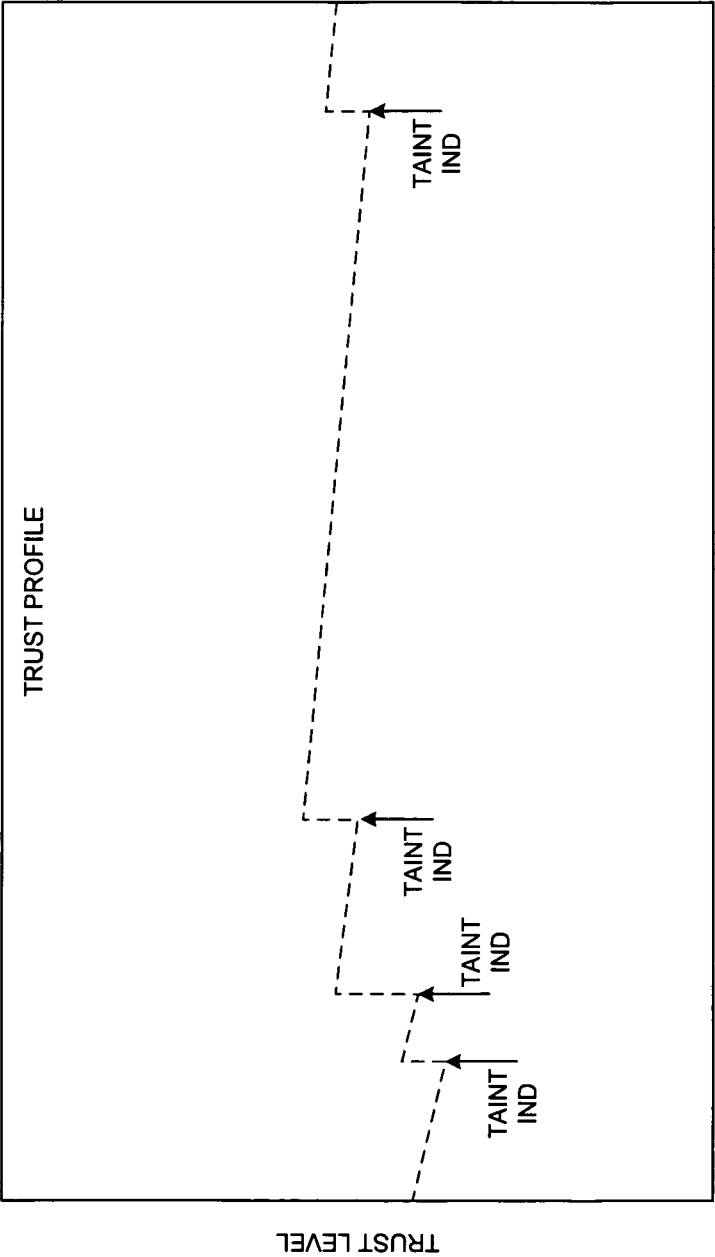


FIG. 1F

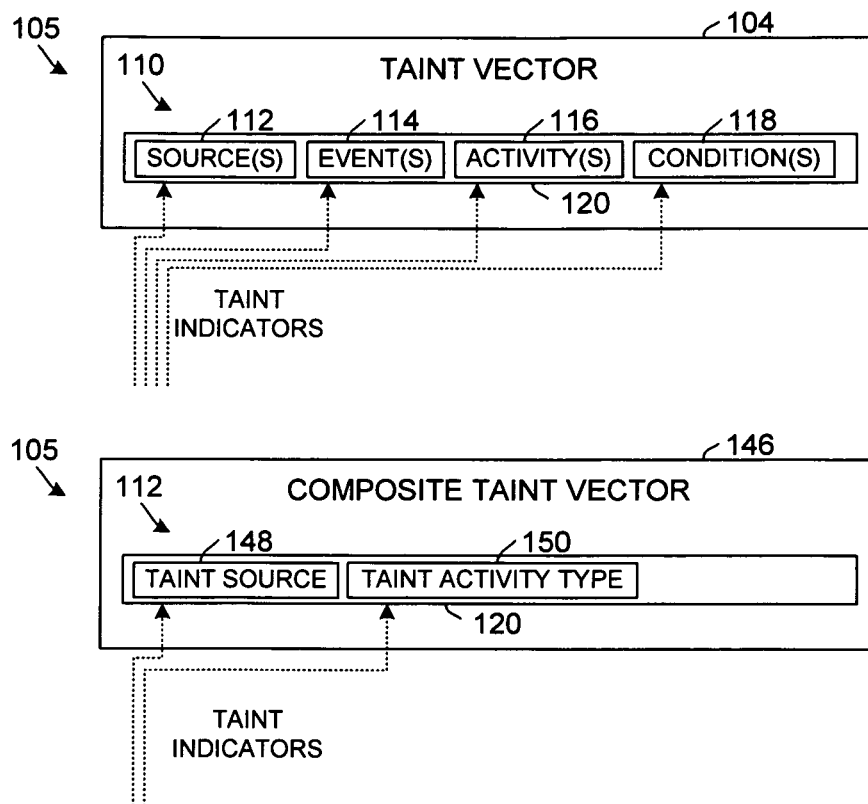


FIG. 1G

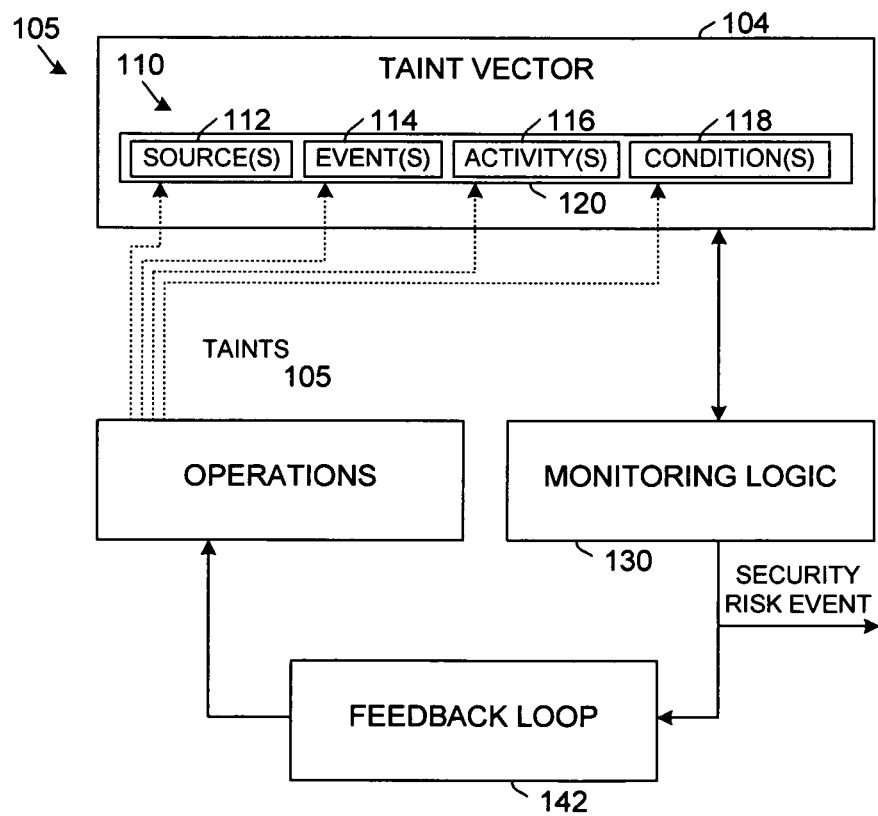
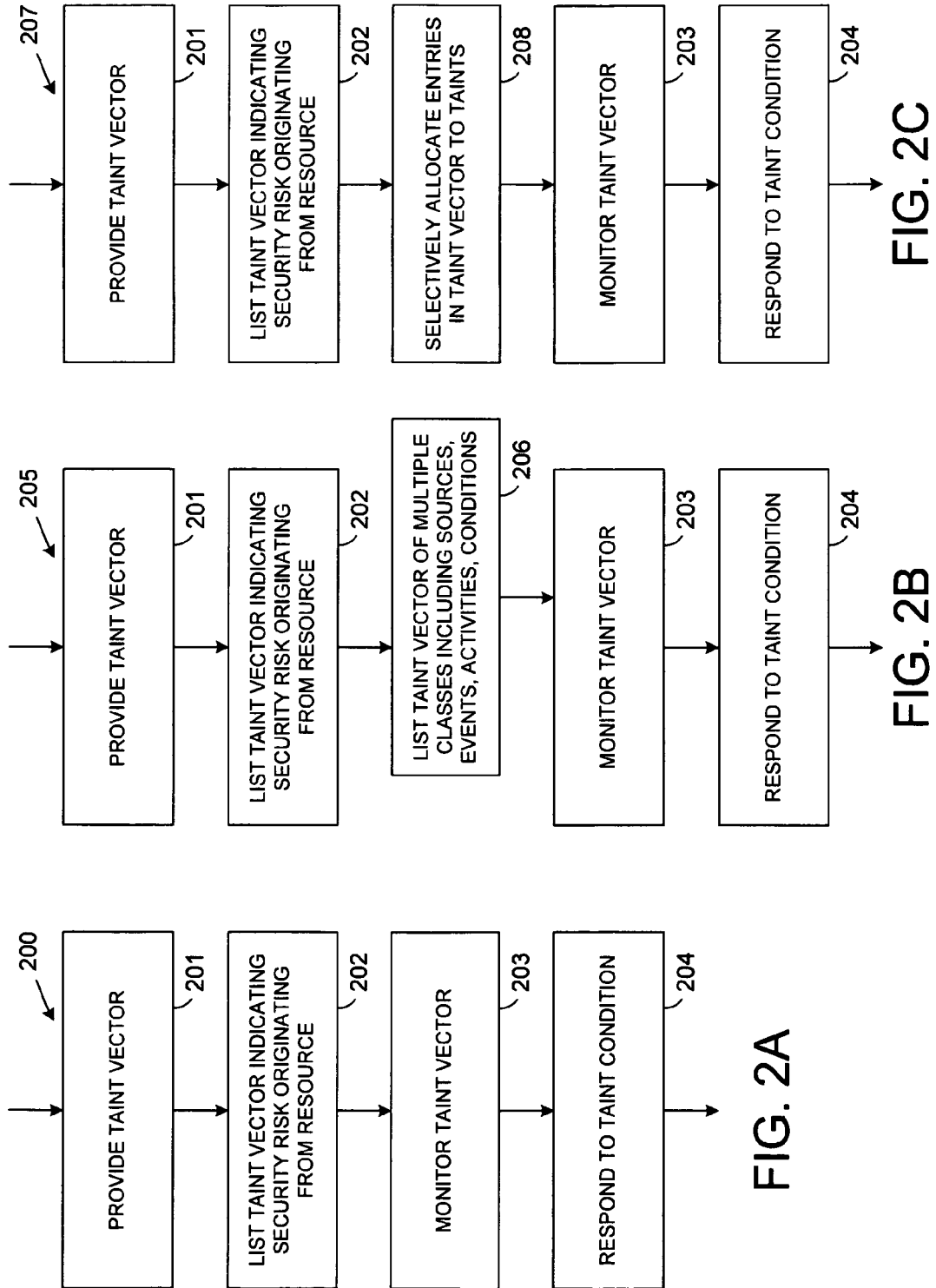
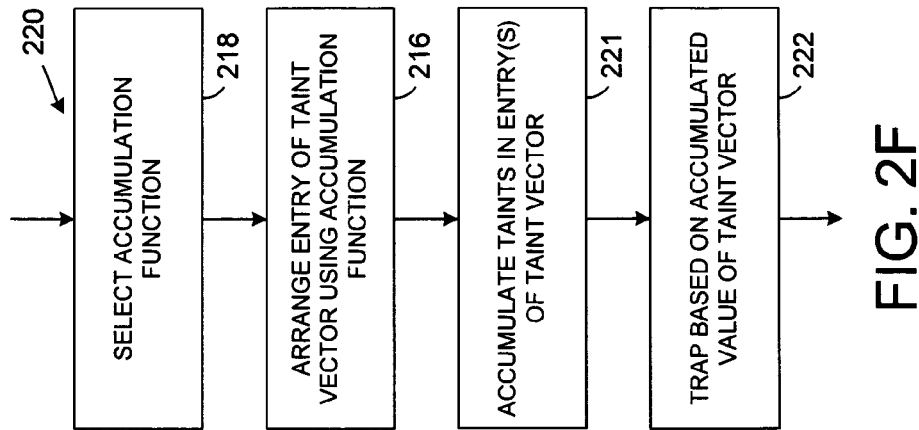
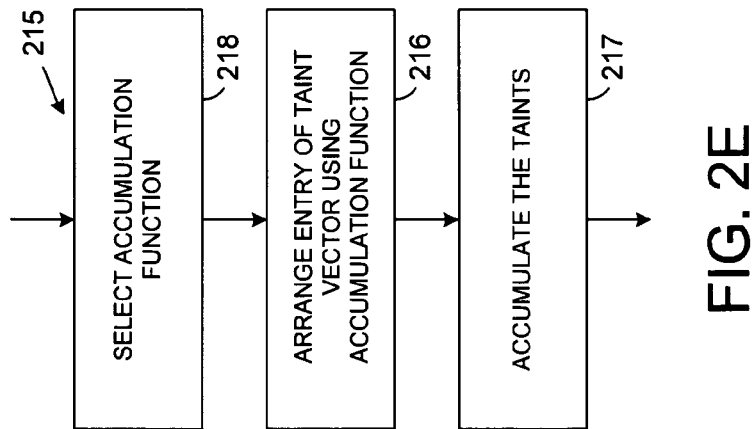
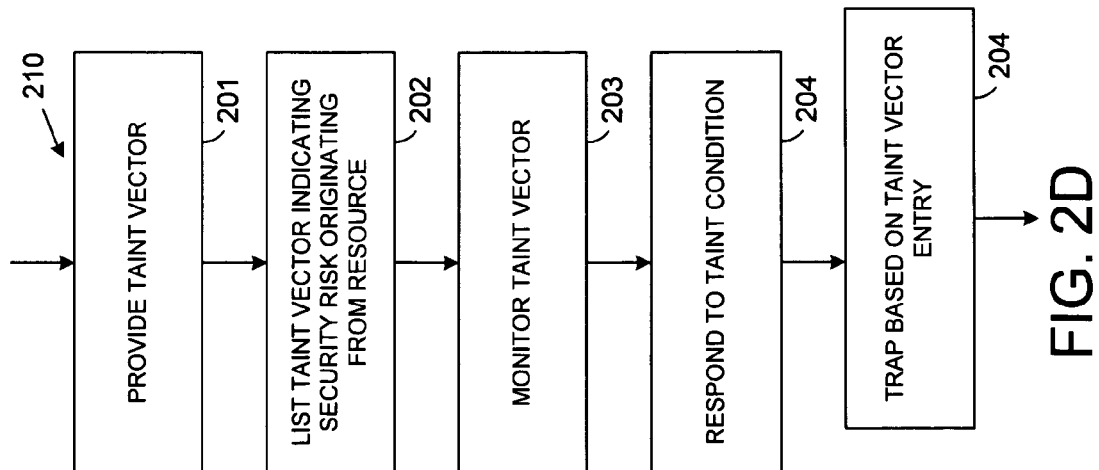
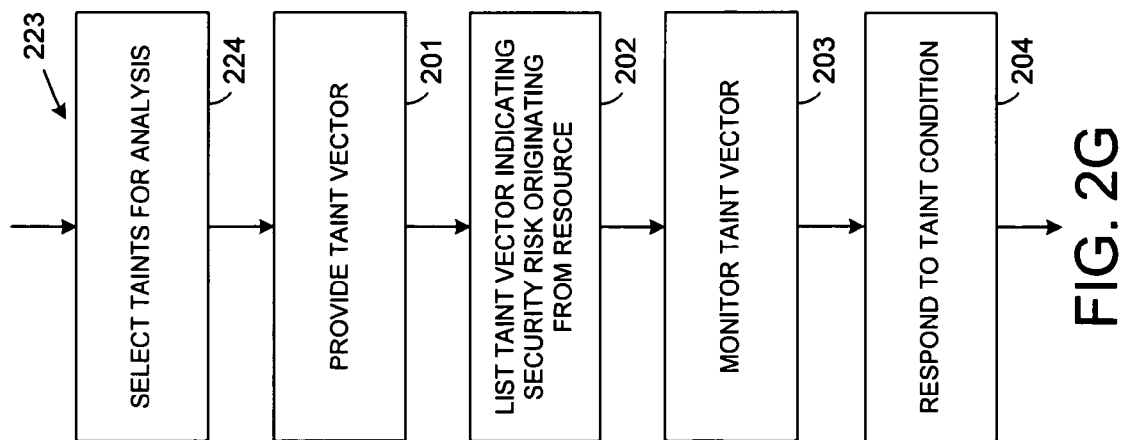
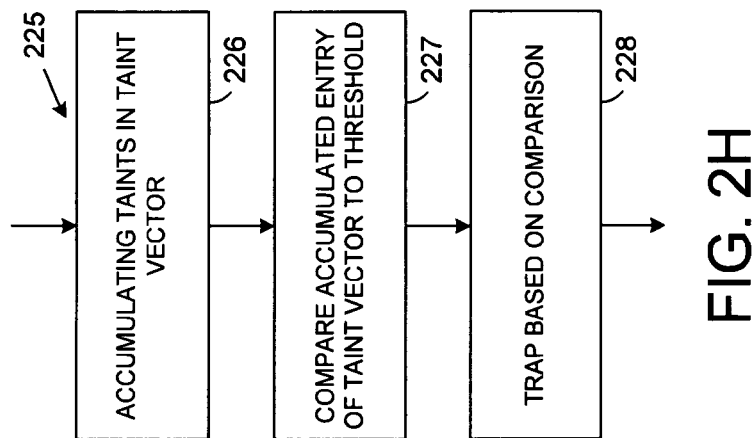
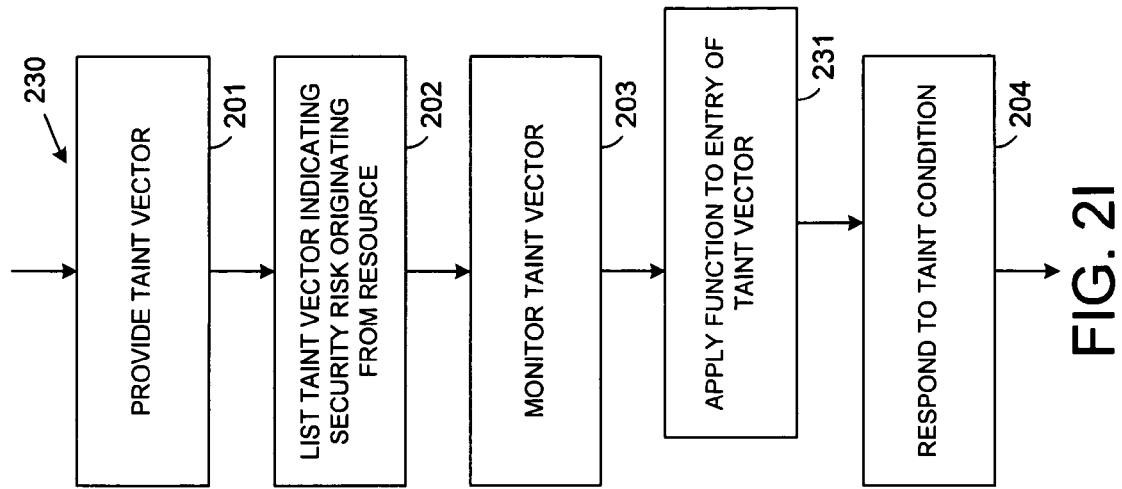


FIG. 1H







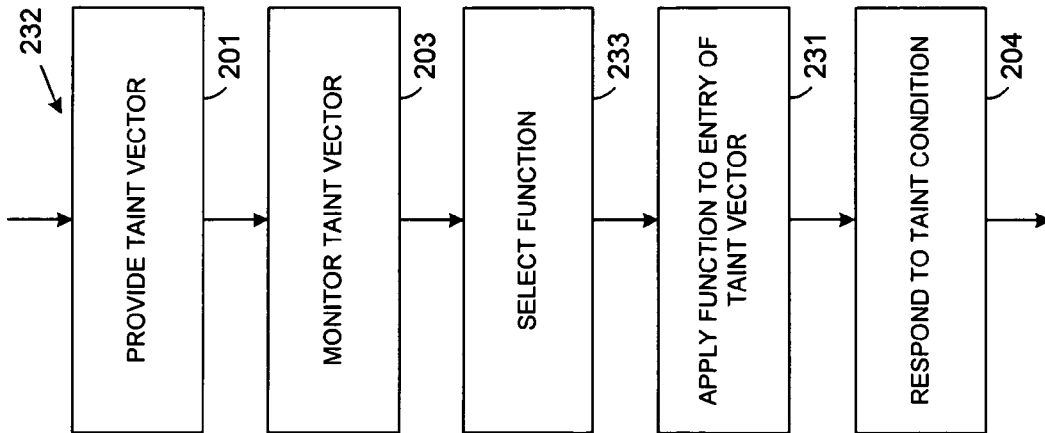


FIG. 2J

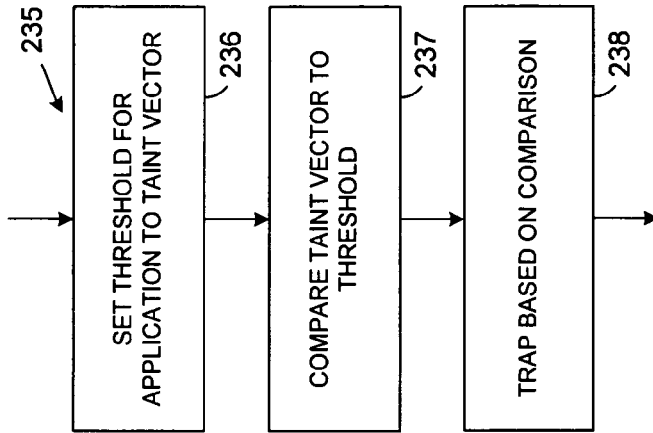


FIG. 2K

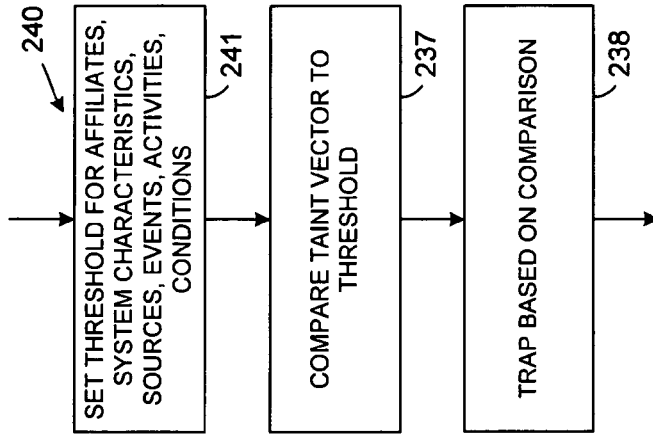


FIG. 2L

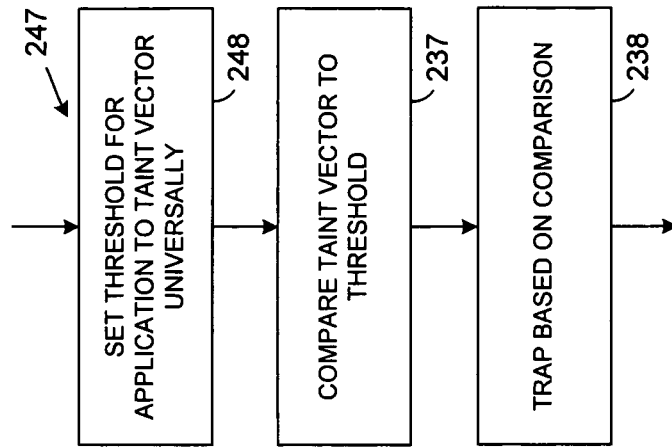


FIG. 20

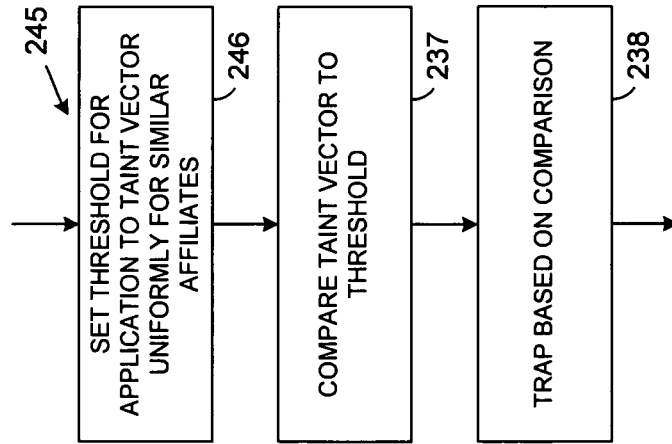


FIG. 2N

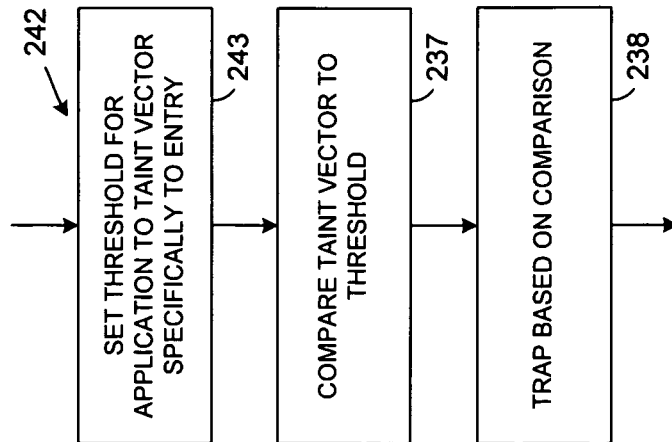


FIG. 2M



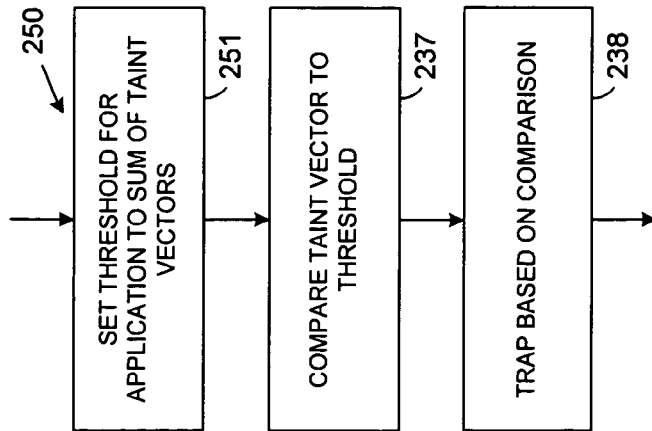


FIG. 2P

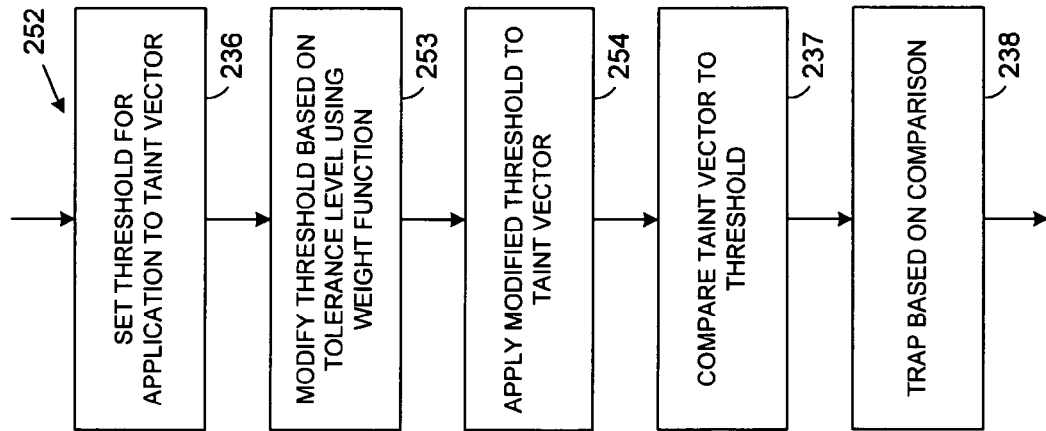


FIG. 2Q

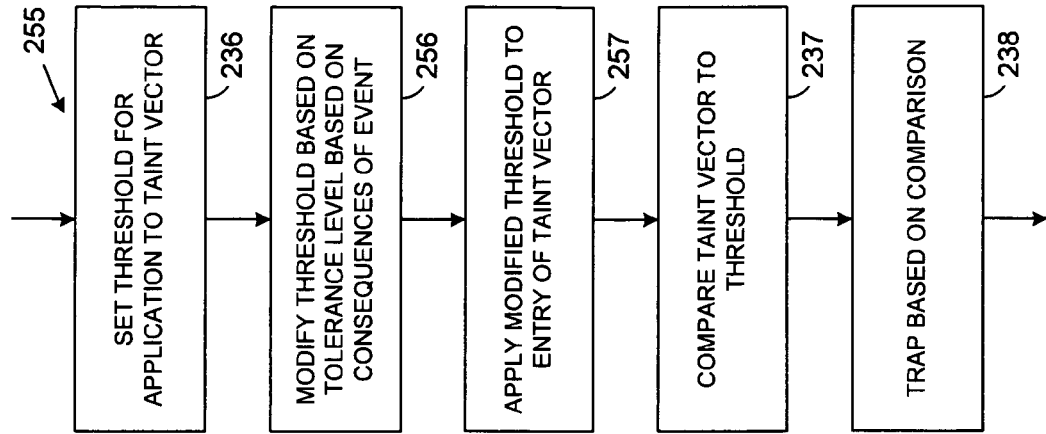


FIG. 2R

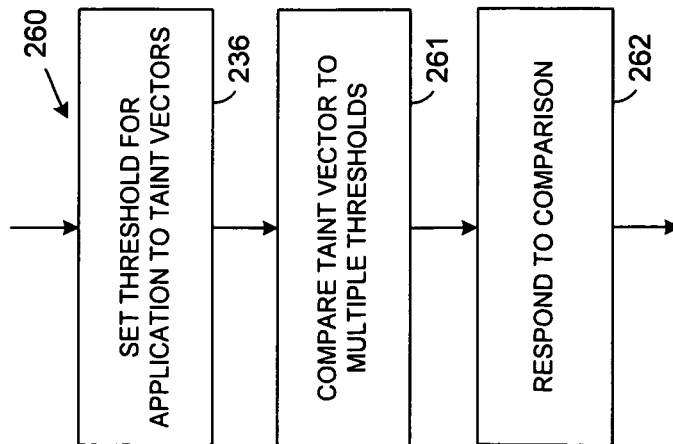


FIG. 2S

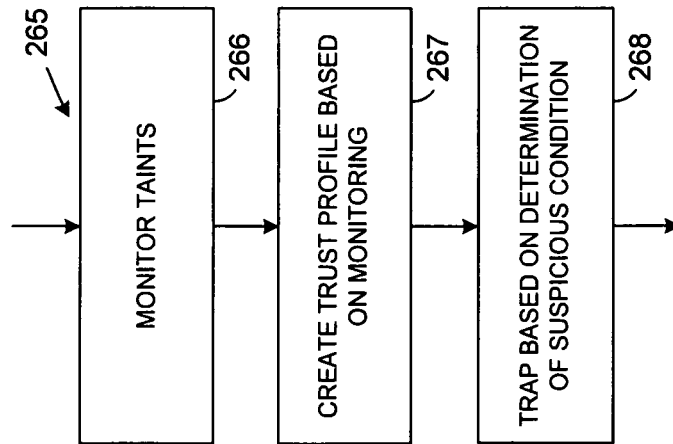


FIG. 2T

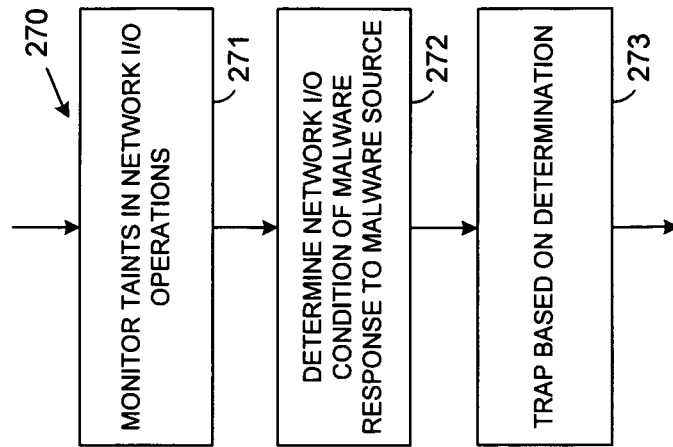


FIG. 2U

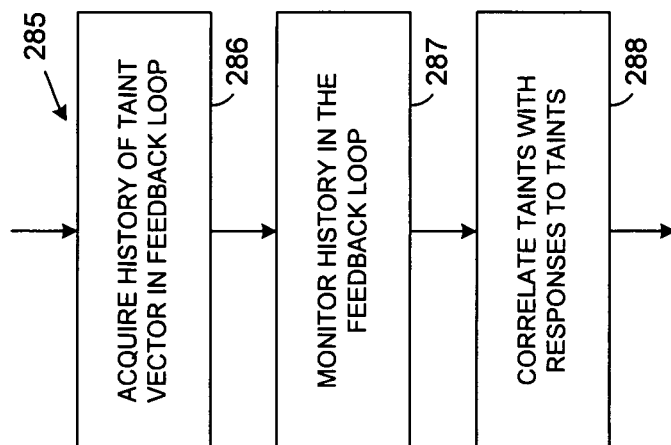


FIG. 2X

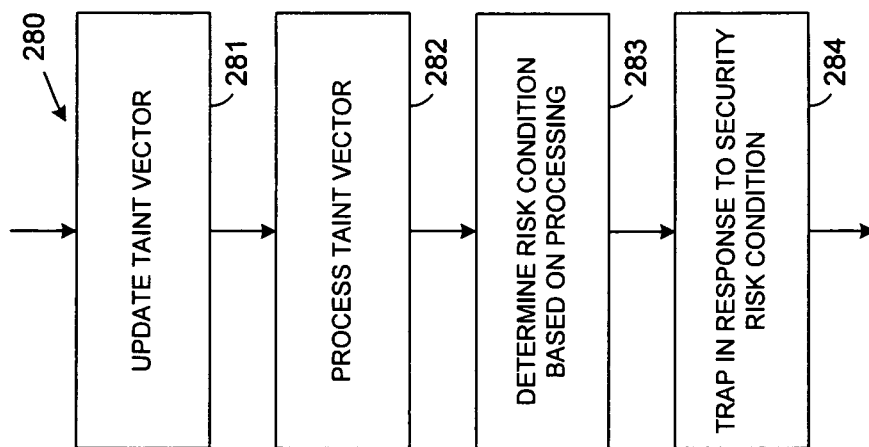


FIG. 2W

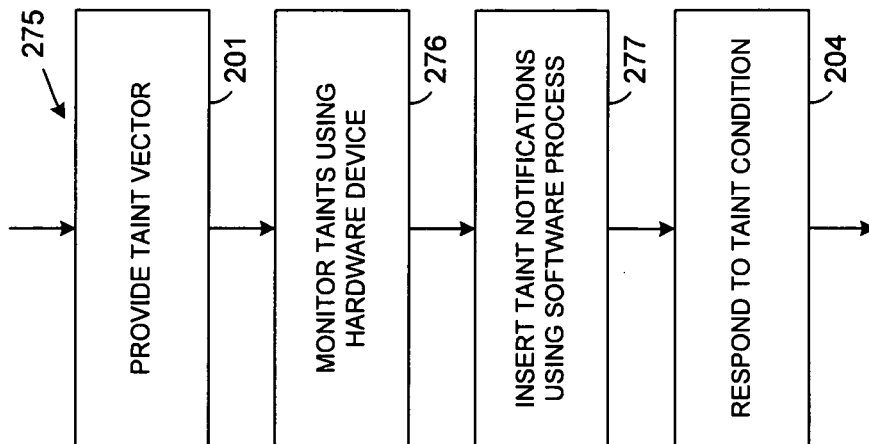
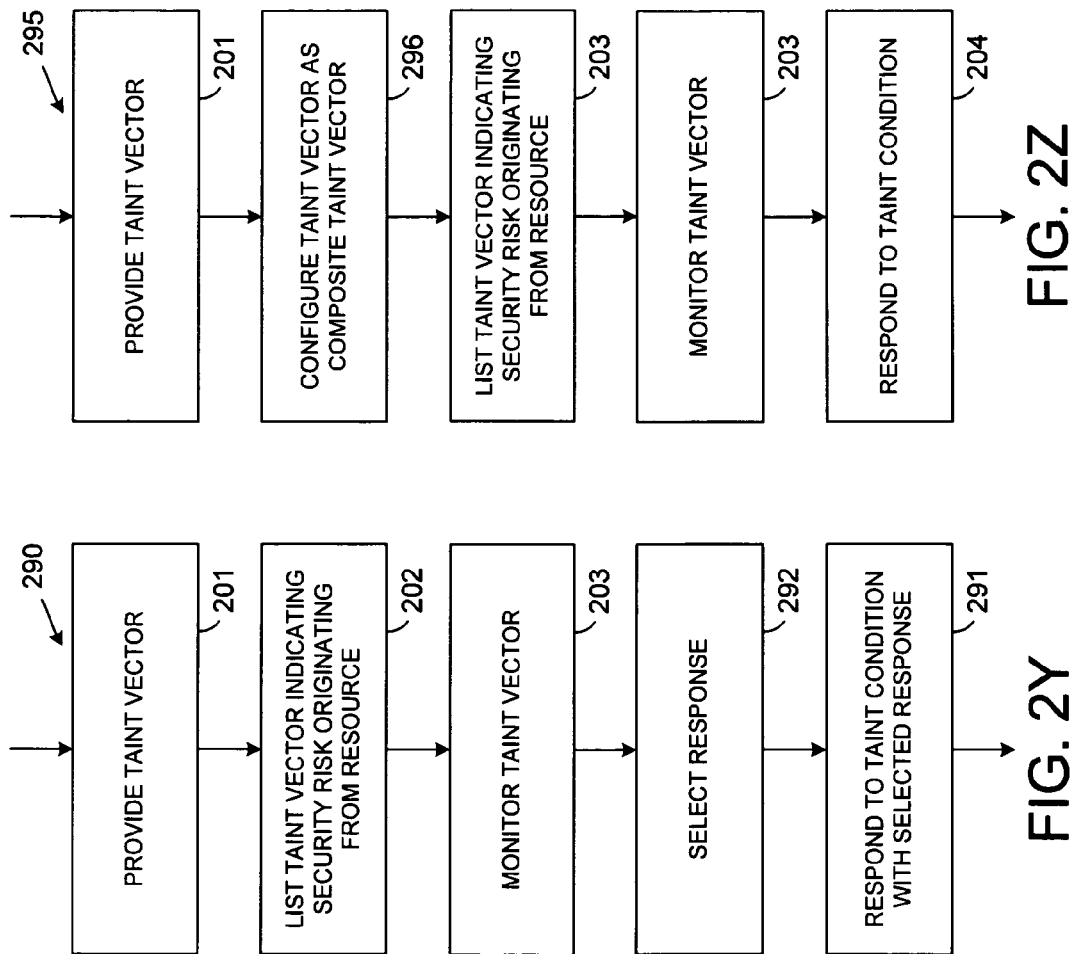


FIG. 2V



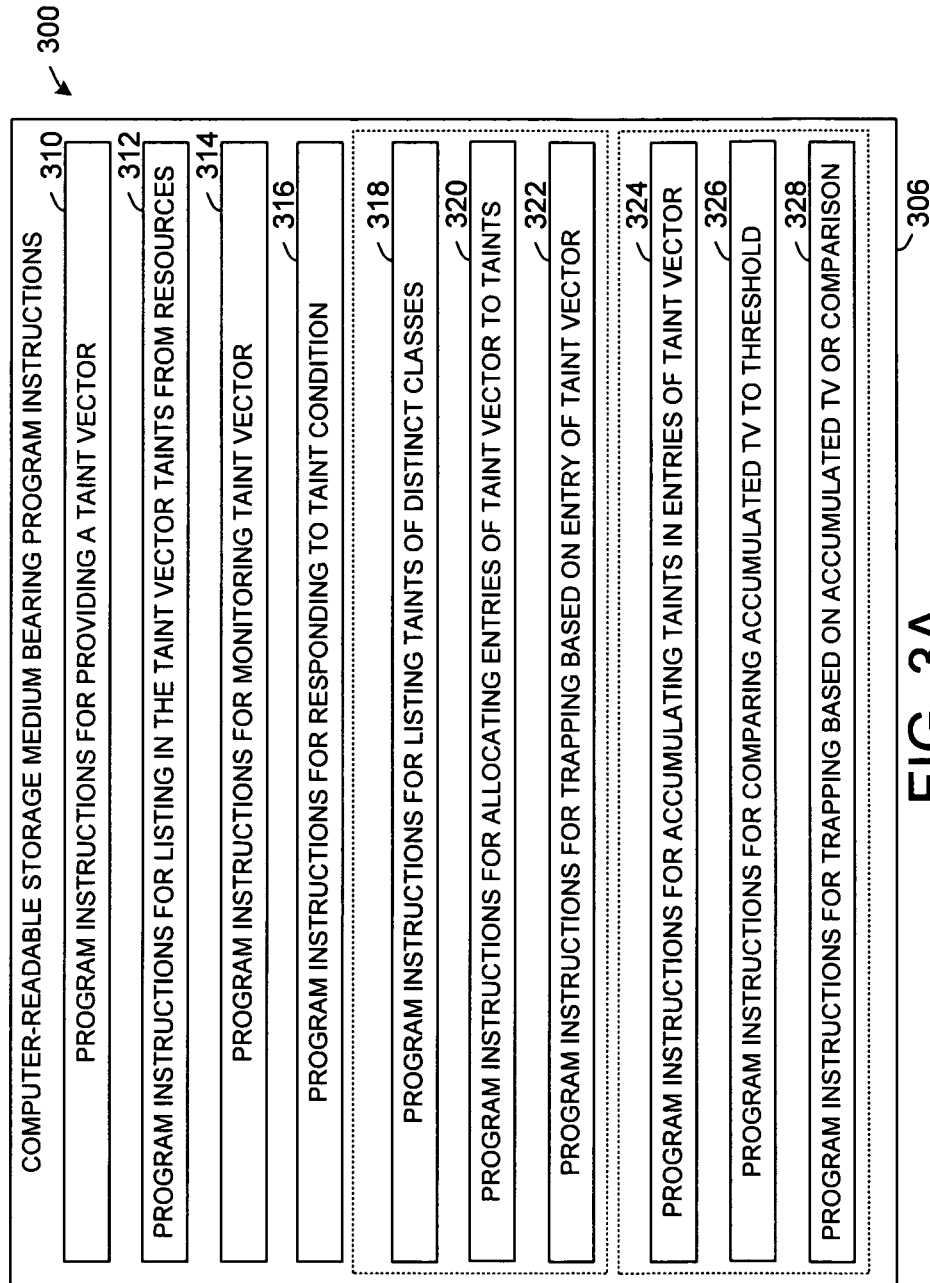


FIG. 3A

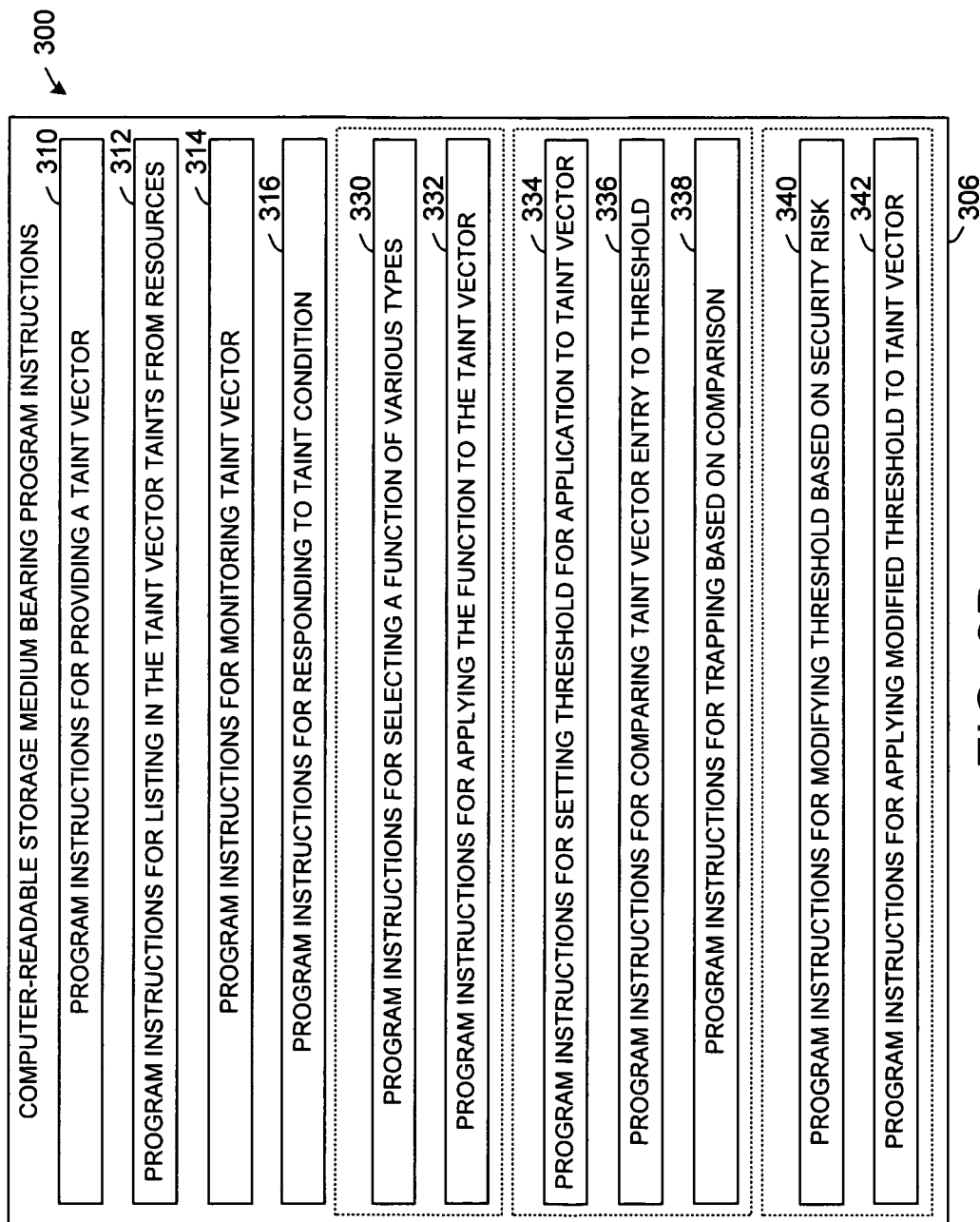
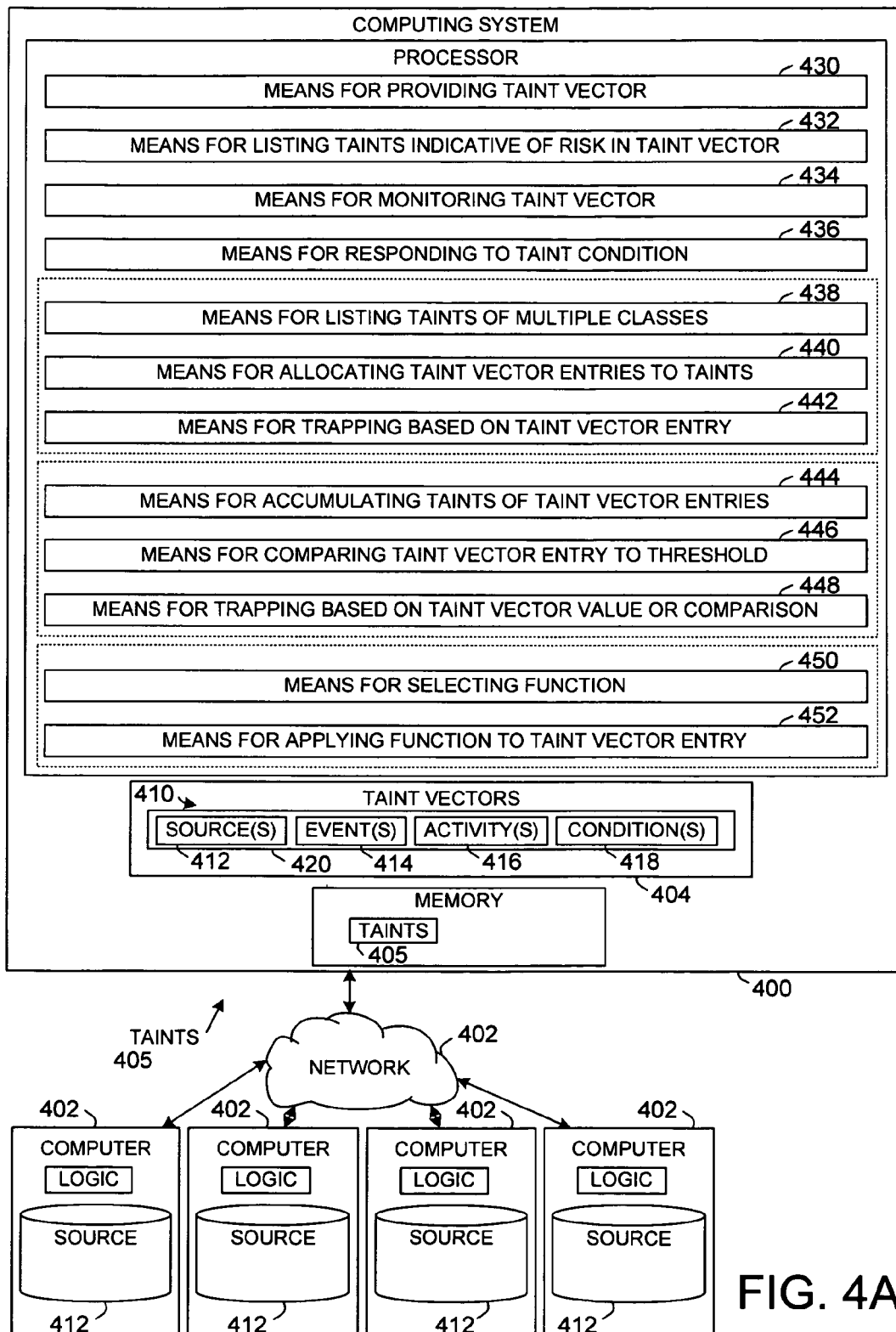
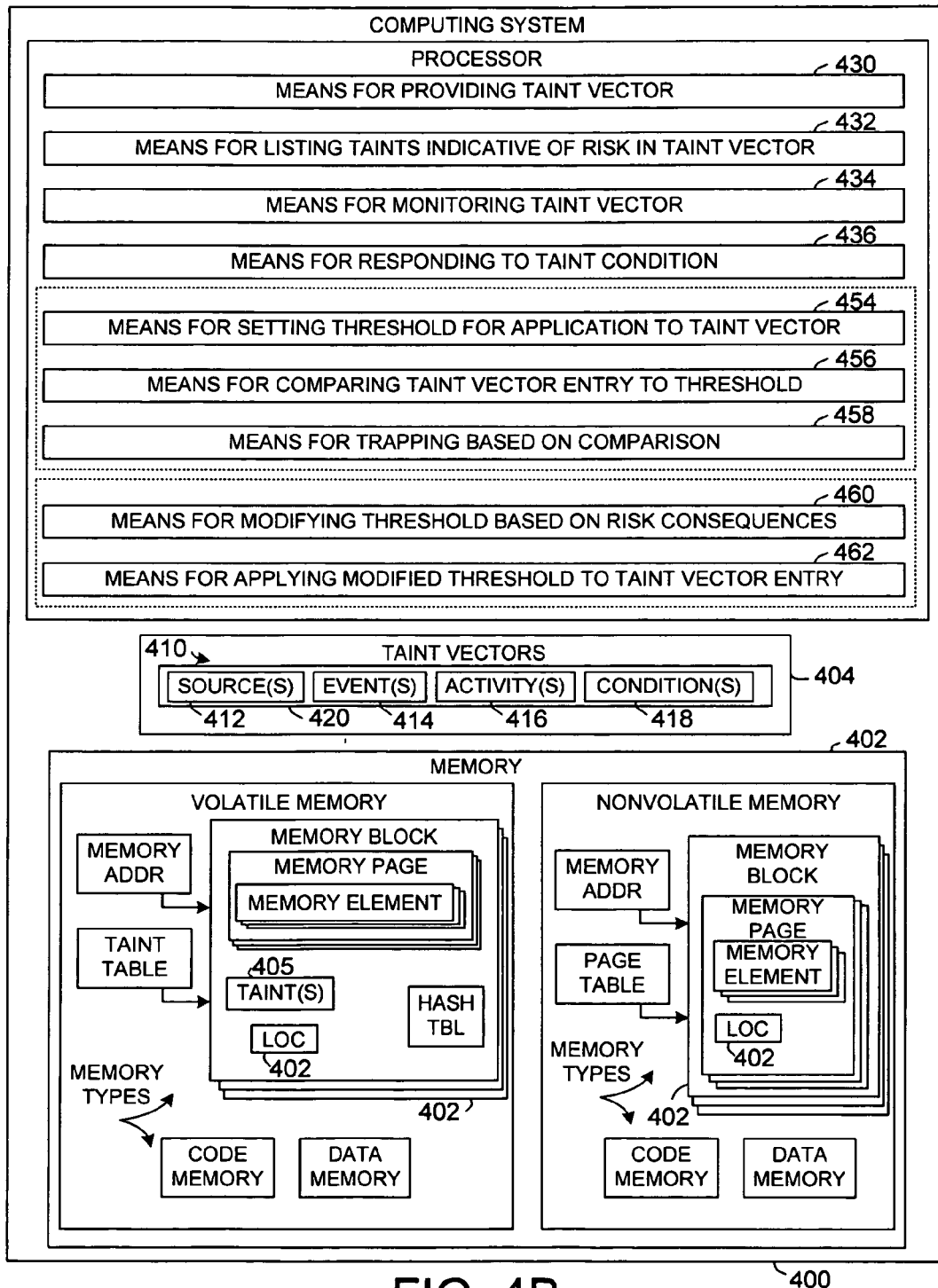


FIG. 3B







1

**CONDITIONAL SECURITY RESPONSE  
USING TAINT VECTOR MONITORING****CROSS REFERENCE TO RELATED  
APPLICATIONS**

The present application is related to and claims the benefit of the earliest available effective filing date(s) from the following listed application(s) (the "Related Applications") (e.g., claims earliest available priority dates for other than provisional patent applications or claims benefits under 35 USC §119(e) for provisional patent applications, for any and all parent, grandparent, great-grandparent, etc. applications of the Related Application(s)).

**RELATED APPLICATIONS**

For purposes of the United States Patent and Trademark Office (USPTO) extra-statutory requirements (described more fully below), the present application is:

1. For purposes of the USPTO extra-statutory requirements, the present application constitutes a continuation-in-part of U.S. patent application Ser. No. 13/136,024 entitled CONTROL FLOW INTEGRITY filed on Jul. 19, 2011, and naming Andrew F. Glew, Daniel A. Gerrity, and Clarence T. Tegreene as inventors, which is currently co-pending, or is an application of which a currently co-pending application is entitled to the benefit of the filing date.
2. For purposes of the USPTO extra-statutory requirements, the present application constitutes a continuation-in-part of U.S. patent application Ser. No. 13/136,400 entitled ENCRYPTED MEMORY filed on Jul. 29, 2011 now U.S. Pat. No. 8,930,714, and naming Andrew F. Glew, Daniel A. Gerrity, and Clarence T. Tegreene as inventors, which is currently co-pending, or is an application of which a currently co-pending application is entitled to the benefit of the filing date.
3. For purposes of the USPTO extra-statutory requirements, the present application constitutes a continuation-in-part of U.S. patent application Ser. No. 13/136,401 entitled FINE-GRAINED SECURITY IN FEDERATED DATA SETS filed on Jul. 29, 2011 now U.S. Pat. No. 8,943,313, and naming Andrew F. Glew, Daniel A. Gerrity, and Clarence T. Tegreene as inventors, which is currently co-pending, or is an application of which a currently co-pending application is entitled to the benefit of the filing date.
4. For purposes of the USPTO extra-statutory requirements, the present application constitutes a continuation-in-part of U.S. patent application Ser. No. 13/136,666 entitled SECURITY PERIMETER filed on Aug. 4, 2011, and naming Andrew F. Glew, Daniel A. Gerrity, and Clarence T. Tegreene as inventors, which is currently co-pending, or is an application of which a currently co-pending application is entitled to the benefit of the filing date.
5. For purposes of the USPTO extra-statutory requirements, the present application constitutes a continuation-in-part of U.S. patent application Ser. No. 13/136,670 entitled PROCESSOR OPERABLE TO ENSURE CODE INTEGRITY filed on Aug. 4, 2011, and naming Andrew F. Glew, Daniel A. Gerrity, and Clarence T. Tegreene as inventors, which is currently co-pending, or is an application of which a currently co-pending application is entitled to the benefit of the filing date.

2

6. For purposes of the USPTO extra-statutory requirements, the present application constitutes a continuation-in-part of U.S. patent application Ser. No. 13/199,368 entitled INTRUSTION DETECTION USING TAINT ACCUMULATION filed on Aug. 26, 2011, and naming Andrew F. Glew, Daniel A. Gerrity, and Clarence T. Tegreene as inventors, which is currently co-pending, or is an application of which a currently co-pending application is entitled to the benefit of the filing date.
7. For purposes of the USPTO extra-statutory requirements, the present application constitutes a continuation-in-part of U.S. patent application Ser. No. 13/200,547 entitled INTRUSTION SET ADAPTED FOR SECURITY RISK MONITORING filed on Sep. 24, 2011 now U.S. Pat. No. 8,955,111, and naming Andrew F. Glew, Daniel A. Gerrity, and Clarence T. Tegreene as inventors, which is currently co-pending, or is an application of which a currently co-pending application is entitled to the benefit of the filing date.
8. For purposes of the USPTO extra-statutory requirements, the present application constitutes a continuation-in-part of U.S. patent application Ser. No. 13/200,557 entitled RESOURCE ALLOCATION USING ENTITLEMENTS filed on Sep. 24, 2011 now U.S. Pat. No. 9,170,843, and naming Andrew F. Glew, Daniel A. Gerrity, and Clarence T. Tegreene as inventors, which is currently co-pending, or is an application of which a currently co-pending application is entitled to the benefit of the filing date.
9. For purposes of the USPTO extra-statutory requirements, the present application constitutes a continuation-in-part of U.S. patent application Ser. No. 13/200,551 entitled RESOURCE ALLOCATION USING A LIBRARY WITH ENTITLEMENT filed on Sep. 24, 2011, and naming Andrew F. Glew, Daniel A. Gerrity, and Clarence T. Tegreene as inventors, which is currently co-pending, or is an application of which a currently co-pending application is entitled to the benefit of the filing date.
10. For purposes of the USPTO extra-statutory requirements, the present application constitutes a continuation-in-part of U.S. patent application Ser. No. 13/200,556 entitled RESOURCE ALLOCATION WITH ENTITLEMENT HINTS filed on Sep. 24, 2011, and naming Andrew F. Glew, Daniel A. Gerrity, and Clarence T. Tegreene as inventors, which is currently co-pending, or is an application of which a currently co-pending application is entitled to the benefit of the filing date.
11. For purposes of the USPTO extra-statutory requirements, the present application constitutes a continuation-in-part of U.S. patent application Ser. No. 13/317,834 entitled ENTITLEMENT VECTOR WITH RESOURCE AND/OR CAPABILITIES FIELDS filed on Oct. 28, 2011 now U.S. Pat. No. 9,098,608, and naming Andrew F. Glew, Daniel A. Gerrity, and Clarence T. Tegreene as inventors, which is currently co-pending, or is an application of which a currently co-pending application is entitled to the benefit of the filing date.
12. For purposes of the USPTO extra-statutory requirements, the present application constitutes a continuation-in-part of U.S. patent application Ser. No. 13/317,826 entitled ENTITLEMENT VECTOR FOR MANAGING RESOURCE ALLOCATION filed on Oct. 28, 2011 now U.S. Pat. No. 8,813,085, and naming

Andrew F. Glew, Daniel A. Gerrity, and Clarence T. Tegreene as inventors, which is currently co-pending, or is an application of which a currently co-pending application is entitled to the benefit of the filing date.

13. For purposes of the USPTO extra-statutory requirements, the present application constitutes a continuation-in-part of U.S. patent application Ser. No. 13/317,825 entitled TAIN T VECTOR LOCATIONS AND GRANULARITY filed on Oct. 28, 2011, and naming Andrew F. Glew, Daniel A. Gerrity, and Clarence T. Tegreene as inventors, which is currently co-pending, or is an application of which a currently co-pending application is entitled to the benefit of the filing date.

The United States Patent and Trademark Office (USPTO) has published a notice to the effect that the USPTO's computer programs require that patent applicants reference both a serial number and indicate whether an application is a continuation or continuation in part. Stephen G. Kunin, *Benefit of Prior-Filed Application*, USPTO Electronic Official Gazette, Mar. 18, 2003. The present applicant entity has provided a specific reference to the application(s) from which priority is being claimed as recited by statute. Applicant entity understands that the statute is unambiguous in its specific reference language and does not require either a serial number or any characterization such as "continuation" or "continuation-in-part." Notwithstanding the foregoing, applicant entity understands that the USPTO's computer programs have certain data entry requirements, and hence applicant entity is designating the present application as a continuation in part of its parent applications, but expressly points out that such designations are not to be construed in any way as any type of commentary and/or admission as to whether or not the present application contains any new matter in addition to the matter of its parent application(s).

All subject matter of the Related Applications and of any and all parent, grandparent, great-grandparent, etc. applications of the Related Applications is incorporated herein by reference to the extent such subject matter is not inconsistent herewith.

## BACKGROUND

Malicious software, also called malware, refers to programming (code, scripts, active content, and other software) designed to disrupt or deny operation, gather information to violate privacy or exploitation, gain unauthorized access to system resources, and enable other abusive behavior. The expression is a general term used by computer professionals to mean a variety of forms of hostile, intrusive, or annoying software or program code.

Malware includes various software including computer viruses, worms, Trojan horses, spyware, dishonest adware, scareware, crimeware, rootkits, and other malicious and unwanted software or program, and is considered to be malware based on the perceived intent of the creator rather than any particular features. In legal terms, malware is sometimes termed as a "computer contaminant," for example in the legal codes of U.S. states such as California.

## SUMMARY

An embodiment or embodiments of a computing system can conditionally trap based on a taint vector. A computing system can comprise at least one taint vector operable to list at least one of a plurality of taints indicative of potential security risk originating from at least one of a plurality of

resources, and response logic operable to monitor the at least one taint vector and respond to a predetermined taint condition.

## BRIEF DESCRIPTION OF THE DRAWINGS

Embodiments of the invention relating to both structure and method of operation may best be understood by referring to the following description and accompanying drawings:

FIGS. 1A, 1B, 1C, 1D, and 1E are respectively, first, second, third, fourth, and fifth schematic block diagrams, and a graphical data description depict embodiments of a computing system adapted to manage security risk by monitoring taint indications and responding to a taint condition indicative of security risk;

FIG. 1F is a graphical data description showing an aspect of operation of the computing system;

FIGS. 1G and 1H are data structure diagrams illustrating example embodiments of taint vector;

FIGS. 2A through 2Z are schematic flow diagrams depicting an embodiment or embodiments of a method operable in a computing device adapted to manage security risk by monitoring taints and responding to predetermined taint conditions detected by the monitoring, for example by conditionally trapping based on the taint vector;

FIGS. 3A and 3B are schematic block diagrams showing embodiments of a computer program product adapted to manage security risk by responding to monitored taint indications; and

FIGS. 4A and 4B are schematic block diagrams illustrating embodiments of a computing system adapted to manage security risk by monitoring taint indications and responding to a detected security risk condition.

## DETAILED DESCRIPTION

In various embodiments, computer systems and associated methods can be configured to include one or more of several improvements that facilitate security. One aspect can be accumulation of taint indicators to distinguish between safe and potentially unsafe data received from safe and potentially unsafe sources. Another aspect is specification and usage of a taint vector to enable monitoring and tracking of a large number of resources and conditions or a wide variety of types without burdening the system and operations with a significant amount of hardware and complexity.

Security in existing networks, systems, and computers is coarse-grained due to large granularity of native code, for example imposed by the 4 kilobyte (kb) size of a virtual memory page. Security is sought in an environment characterized by running of applications that share data with other entities. Security is coarse-grained in that memory blocks can be individually protected. For binary code or machine code, the 4 kb granularity encompasses a large amount of data in comparison to the typical 10 or 12-bit size of machine code words for which individual protection may be sought.

Another security technique can be to assign data to a particular virtual machine, which is even more coarse-grained. For example, if security is sought in the context of a browser not known to be secure, the browser can be assigned a virtual machine that runs only the browser. A virtual machine can encompass more than a CPU alone and include other components and devices such as motherboard I/O devices. The virtual machine thus can be much larger than the 4 kb granularity of memory blocks.

Security can also be sought in software or interpretive environments, for example using Java byte code or C-sharp byte code, which can be more fine-grained but at the cost of much slower performance. An interpreter can support any protection desired, even down to individual bits but is much slower than the machine code level. Performance can be accelerated only by more coarse-grained checking.

What is desired is fine-grained security with suitable speed performance. Fine-grained security is directed toward protecting memory in fine-grained pieces.

Fine-grained security can support resource allocation and resource scheduling, and can be supporting technology for hardware scheduling, virtual memory. Fine-grained security facilitates, for example, for running applications on a computer controlled and owned by another entity.

Various techniques can be used to identify the memory items to be protected including pointers such as a pointer to an object or metadata associated with a pointer, offsets, addresses, and the like.

An example fine-grained security paradigm can use metadata associated with a pointer that identifies a lower bound, and upper bound, and permissions. The pointer can be enabled to point to particular objects or even to any position within an object. Metadata can specify permissions including memory locations to which data can be written, when program code is allowed to execute, how long writing is allowed, and the like. Permissions can be associated with data objects, for example assigning a pointer to an object and, using permissions, allowing only methods belonging to that object to access the object. Another example of permissions can enable access to data, but only for specified purposes, for instance to enable a first running of an object and access allowed to only part of a routine, while preventing access by others. In another example, a particular method can be permitted to run a limited number of times or just one time, and can prevent subsequent access to data when the data has been previously exposed to an authorized reader.

Permissions can implement a concept of poisoning. For example, a user can enter a name into a text field and mark a poisoned bit that prevents subsequent branching or sub-routine return. The poisoned bit can function as a dirty bit which indicates whether an item such as an object, memory, or other resource is dirty, which prevents predetermined purposes or actions to the item, for example preventing actions applied to a data block or object, such as not allowing return.

An illustrative computer system can be configured for fine-grained security as supporting infrastructure in a concept of federated sharing and federated data sets. Sensor fusion involves fusing of data and data sets in numerical aspects and permissions aspects, wherein data and data sets are fused in conditions of a first entity owning or controlling a first sensor and a second entity a second sensor.

Fine-grained security can be implemented in an infrastructure can be implemented in an architecture including servers and clients. For example, gaming code servers and gaming console clients can interact by running program code that executes in part on machines controlled by the server and in part on machines controlled by the client. Fine-grained security enables the interaction to be mutually trusted by both sides.

Fine-grained security can be configured to exploit existing infrastructure aspects such as the Trusted Platform Module (TPM) which is installed in computer systems somewhat universally but little used in practice. TPM generally includes secure storage for keys little or no security logic.

In some embodiments, a servers and clients architecture can implement fine-grained security using one or more server downloaded modules. For example, a gaming code server can transfer a server downloaded module that executes on a client wherein the client's user software and operating system is not able to read associated TPM keys. Fine-grained security can be configured to prevent the client or user operating system from reading the TPM keys, for example to ensure isolation in software, and further configured to prevent physical attacks for example via a device such as a logic analyzer on the bus reading sensitive information.

Some system embodiments which support fine-grained security can be activated at boot-strap loading of a computer, for example via microcode executing in the processor. A further aspect of fine-grained security can include physical security of the TPM, for example through use of tamper-evident/resistant packaging. At boot-strap loading, TPM can perform various security operations such as inspecting software version and possibly microcode, ensuring viability of software, for example by creating and applying a hash to each level of code (microcode, firmware, software, and the like), checking against previously run code, signing-off on viability if warranted, and printing a signature of executing code to enable determination of trust.

Fine-grained security operations can further include building or creating a chain of trust, checking each part of operation beginning with TPM, then checking security during operating system functions, downloading of modules, and execution of procedures. In an example configuration, fine-grained security can perform checks of operation system functions which, to the first order, control all operations.

An example of chain of trust can begin with trust of an operating system (for example by an association such as Motion Picture Association of America (MPAA), International Game Developers Association (IGDA), and the like). If the operating system is certified and fine-grained security operable under the certified operating system ensures that the system is not hacked, the chain of trust is established since the operating system prevents user code from accessing downloadable code.

Weaknesses of the chain of trust can be that the process is too linear and easy to break since a single-point of failure breaks trust. Chain of trust also has problems ensuring privacy.

An extension that can improve chain of trust is a late-secure boot which is run later than a typical bootstrap load and can involve security checking in an operating system that is not yet trusted. At running of the late-secure boot, a security initialization is run which starts security process booting in a system that is already running.

A more secure concept of security can be a web of trust. The web of trust can have multiple trust levels which hand trust to the operating system. At each trust level, software can validate code in a stack of code to establish trust. In the web of trust, a failure at some point can be rechecked according to a byzantine path which forms a set of protocols to establish trust. The operating system can use pathfinding or agglomerated trust protocols to analyze trust at each level to enable multiple levels or types of trust validation.

Intrusion detection can be an aspect of fine-grained security.

Intrusion detection can use the concept of poisoning to implement fine-grained security. Poisoning can be used for protection, for example in the case of sensor data or a sensor controlled by an untrusted entity. One or more bits can be

allocated to identify aspects of the target sensor and data. Poisoning can be data-defined or entity-defined.

A system can enforce security via accumulation which can be used to quantify poisoning, for example by accumulating multiple indicators of lack of safety or “dirtiness.” Accumulation can be operable to accumulate on a per-location basis, per-data basis, overall, or any selected combination. Accumulation can be used to quantify whether data from a particular source or entity can be trusted, rather than to detect security attacks per se.

A taint technique can be used to distinguish between safe and potentially unsafe data received from safe and potentially unsafe sources. The term “taint” can be defined as potentially unsafe data or data received from a potentially unsafe source. Unsafe data and/or sources are untrusted as potentially dangerous, malicious, or suspect according to a predetermined security policy. Security criteria of tainting can be specified independently for various applications, conditions, and/or implementations ranging, for example, from a source, data, and/or resources via which the data is transmitted that are not known to be completely trusted to those known to have positive confirmation of ill-intent, malice, or compromised security attributes. In some implementations, analysis of the data itself may contribute to taint characterization.

Accumulation enables analysis of a particular sensor which is not untrusted as fundamentally faulty or inherently dishonest but rather imperfect to some degree, for example with a signal to noise ratio that allows some errors. Thus, data may be trusted overall or over time, but possibly an individual bit may not be trusted. Accumulators can gather taints up to a predetermined threshold, after which an action may be taken. A taint can arise from software, can be forwarded from an original source, may result from an attacker attempting to break into a web browser, or may be “operational” for null pointers, buffer overruns, and other faults. In various embodiments and/or conditions, accumulation may be per source, overall, or both. One or more bits can be accumulated per untrusted source. The accumulation can be configured to be subject to various selected algorithms, for example power law, race functions, and the like.

In a power law algorithm, the frequency of a security risk event is presumed to vary as a power of some attribute of the event. The power law relationship is believed to apply to distributions of a wide variety of physical, biological, and man-made phenomena such as sizes of geophysical and weather events, neuronal activity patterns, frequencies of words in various languages, and many other examples.

In a race function, a security risk event is presumed to follow exponential or geometric change, either growth or decay, wherein the rate of change of a mathematical function is proportional to the function’s current value.

An accumulator can be configured using any suitable arithmetic or logic element, and can accumulate data in any suitable manner, such as a counter or bit per source, a bit per accumulator. The accumulator can be configured to address information from different sources and at different times in selected distinctive manners. For example, an accumulator can be set so that 99% correct data is sufficient and a clean bit indicated despite occasional errors, while data from another source may be known to be valid only 65% of the time wherein a selected algorithm can be run, for example power law, race function, or the like, to determine validity.

On the specific case of sensor, some errors occur because sensors aren’t perfect, a signal to noise characteristic is present so some errors will occur, even in the case that data is usually correct 99% of the time. Thus, the data can be

generally trusted cumulatively with some level of trust to individual bits. An entity that is not trusted will have outlier in terms of error rate, not criteria per error rates. In some circumstances one definition of trusted/untrusted can be specified or tracking can be done on source and data basis. In a federated system, tracking can be on the basis of the sensor of one entity against another entity.

Various other accumulator examples can be implemented. A counter per affiliation can be defined wherein a low level is merged up to a higher level. Pathways to a system can track sources of data through a system such as by running data through a specified pathway through a “validator,” a software program or hardware logic used to check the validity of multiple taint indicators in terms of security risk. A 2-4-bit counter can be used to track one-bit per source or a counter per source.

Tainting can be performed on a one-bit basis for a small number of sources which can be federated down to whatever sources are desired. An accumulator can be configured to count the number of taints, such as the taints per memory unit (per byte for example). Statistics can be performed on any suitable taint counter—a counter per bit, 2-bit counter, 4-bit counter and the like. Examples of taints and/or events to filter can be used for taint monitoring and creation of a trust profile and include: instructions tainted, the number of tainted instructions, the number of instructions written as a result, the number of data loads and stores, the number of data memory accesses, outputs, calls/returns, branches (for control flow), integer overflows, network I/O, and the like. An integer overflow can be handled as a taint. Integer overflows occur frequently and can be legitimate about half the time, and thus a condition indicating possible error but by no means certainty of error.

Monitoring of network I/O is useful for detecting when a virus attempts to call home. The system can trap to software if any specified taint occurs, a simple reaction for any suspicious event.

Accumulators can be used to build a trust profile over time, such as by using taint information as raw data for creating the trust profile. The trust profile can be used to lower and raise the trust level over time, and to make subsequent decisions. For example, a bit or counter can decay over time to balance race with accumulation.

Any suitable comparisons can be defined for particular conditions. In an illustration, a trust profile of an I/O process can be built over time. In a simple control scheme, a high-risk operation can be monitored so that if the number of taints is greater than a predetermined threshold, I/O can be blocked. Over time, the count can be decremented to account for spurious events.

Suspicious activities can be monitored using comparisons, for example using a counter or a single-bit designating suspicious events. Examples of suspicious activities can include null pointer references which are not always intentional or malware, buffer overruns/overflows which are usually untrusted, repeated attempts to access a key, and the like.

Comparisons can be used to efficiently track suspicious activities, particularly in conditions that complex statistical analysis is unavailable or unwarranted.

A taint vector, operable as an intrusion detection system, can be created for tracking multiple events or conditions. An example taint vector can comprise 16-64 bits corresponding to associated sources, events, conditions, and/or suspicious activities. Each taint vector of a composite vector may correspond to a source of data or a type of activity. Taint vectors enable monitoring and tracking of a large number of

resources and conditions or a wide variety of types without burdening the system and operations with a significant amount of hardware and complexity. The taint vector can include a various decay options tailored to the particular information monitored. For example, the taint vector can decay after a certain number of operations to avoid triggering on outlying events. Possibly schemes for implementing decay can include: 1) increment/decrement using a single vector which is incrementing and decrementing is performed on the same vector, 2) copying the vector to memory periodically to maintain on old version while continuously incrementing and decrementing to enable restoration of the old version subsequent to reacting to an invalid or error condition, and 3) impose a decay that is a race of decay versus accumulation.

A taint vector can be configured to introduce a new class or type of security element, not taints but rather suspicious activities including null pointers and buffer overflows. Suspicious events are taints or can be treated and propagated like taints.

The taint vector can be tailored to monitor various comparisons including, for example: are any elements greater than threshold, are all greater than threshold, is the sum of all or some elements greater than threshold, is the sum greater than an intermediate value, and the like. The system can trap if the taint vector meets predetermined conditions.

The taint vector can be considered an accumulator of faux pas, for example null pointer references, attempts to access a secure part of the CPU, buffer overruns (a common hacking technique). The taint vector can be used to monitor events or conditions that are not necessarily attacks or failures but may be innocent or coincidental, but originates in a region that raises suspicion, wherein a feature of the region can raise or lower suspicion. A taint vector can be configured to focus more on the type rather than origin of malicious event or condition that occurs. The taint vector can include primary and secondary criteria, and accumulates suspicious actions while also considering indicial of levels of suspiciousness including extra data and extra identifiers relating to the actions for further analysis. Accordingly, although the taint vector can consider the source of an event or condition in determining suspiciousness, actions, consequences, and usage can be more pertinent to identification of an attack or malicious condition. For example, known system calls are associated with reading data off web pages and thus tagged as suspicious for further analysis in which the source of the system calls can be identified (for example via operating system software that injects a label identifying the source).

The taint vector can be configured to set a hierarchy of suspicion based on the source, type, or identify of an event. For example, a buffer overrun can be considered worse than a null reference. The source of the event can be considered to assign a level of suspicion such as whether the sensor from a known and trusted bank or an unknown bank or foreign hack site.

Information can reach the taint vector from multiple various sources. For example, some system calls are associated with accessing information from web pages. These calls are tagged and the operating system injects a label indicating that the data originated from a web browser at a particular identified site. The protocol for receiving a taint notice for tainting originating in a remote system outside the system which controls the taint vector can be that the taint notice is placed by software as some level, possibly software in the remote system. The taint notice is received from software from various sources such as by forwarding from

the originating source, determined by a person attempting to write to a web browser, originating from suspicious operations or faults (such as buffer overflows), and, generally, from an indication that data has some level of questionability.

The taint vector can be implemented to include tolerances set based on questionability of the source or event. Zero tolerance can be set for particularly suspicious or harmful events and/or sources wherein a single event can result in a maximum response. For a low threshold, the response for one taint can result in a trap, exception, or shutdown, and may be used, for example, in nuclear power plant control.

A medium threshold can be a hybrid of low and high threshold and call for a medium response and include aspects of decay. An illustrative setting for medium threshold may allow two taints per hour and thus have decay of one taint per half hour. In a typical condition such as one buffer overflow per X amount of real time or CPU time or other interval, a monitor tracks events. Decay is implemented to account for rare and spurious events that are likely to occur by chance when monitoring continuously for vast time spans, and do not surpass threshold for an error condition. Decay is thus imposed upon accumulation so triggering occurs when more events per unit time (other interval, instruction cycles, and the like) than accommodated by decay are indicative of an error condition. If events occur too often, the threshold of rate of occurrences indicative of suspiciousness (taint rate) is too high and the threshold can be reset.

An example of high threshold can allow twelve taint counts per unit time such as for cheap video forwarded from a provider or signals from ubiquitous cell phones. Most events can be ignored in the absence of some indication of attack. Thresholds are set to balance a sufficient level of security with communications characterized by large amounts of data and frequent errors.

If taints exceed the threshold, then suspicion if sufficiently great that some action or response is taken. A suitable response can be trap, exception, notification, alarms, and the like.

In various system embodiments, taint vectors can be configured at selected locations and with selected granularity. A simple system can have a single taint bit. A slightly more complex system can have a single taint vector allocating multiple entries. Additional control and functionality can be attained by assigning a taint vector per register, for example to track computer processor register EAX (in 32-bit Intel Architecture IA-32) while not tracking register EBX.

A taint vector can be configured to track memory taints, for example tracking every byte in a computationally and resource expensive arrangement. In contrast, a less extensive implementation can assign a single taint for all memory such as with 64 entries. A vector of 64 entries may have one bad indicator operable as a running gauge of operations. The taint vector can indicate on/off status or a range.

Taints can be allocated by memory page which can be challenging for usage with Intel processors since no free bits are available and page tables are read-only. To address this challenge, a system can include a memory taint hash table which, if read-only, can indicate a level of taint per memory block. A read-only memory prevents logging of taints in memory so that the table is located outside of the read-only memory. The amount of memory for the table can be reduced by using a hash. Memory at the hash of an address can be used to compress the address, for example 4 gigabytes (GB) can compress to a 64-kb table. A special instruc-

tion can be specified in which store memory at a specified address receives a predetermined value.

Taints can be allocated by byte to attain the finest possible granularity. A special instruction can be specified in which memory at a specified address has a taint field equal to a predetermined taint field value. Another example special instruction can be specified to create a taint hash vector in which memory receives a specified hash of the address where the hash is operable to compress the address, for example 4-GB of memory can be compressed to a 64-kb table. Once the hash is performed, security is no longer determinant or precise so that false positives can occur. The false positives can be addressed using intrusion detection capabilities of the system. The taint hash vector is costly in terms of resources, possibly 1-2 bits per byte maximum—a substantial amount of overhead.

A taint vector can be configured to segregate memory by type, for example distinguishing memory for storing program code from memory for storing data. Different types of segments can be allocated for corresponding different granularities of taint information.

Taints can be allocated by hardware process identifier (PID). For example, one vector can be allocated per hardware thread to address context switching wherein a software thread's vector is stored.

In another option, taints can be allocated wherein a cross-thread taint is enabled, for example to address some system-wide taint.

In various embodiments, the operation of tainting can be allocated among hardware devices and components and software. In a particular embodiment, hardware can track taints while software can inject initial taint notifications, except when hardware can determine a priori that an event or operation is bad. In example functionality, hardware can generate a trap to software according to predetermined "trap-it" rules that are most suitable selected so that rules are simple and easy to describe, are activated in response to a specific condition, and easy to implement. A trap can be activated based on selected threshold conditions.

In various system embodiments, taint vectors can be configured with selected decay and using selected decay mechanisms. Decay can be applied periodically for example either on a consistent basis or with a varying period based on a sensitivity meter. Characteristics of the sensitivity meter such as rate of subtraction can be selected based on the environment of a system, for example whether a computer is running on a home network, a business environment, a public network, and the like.

Decay methods can include subtraction of selected number N or shifting the taint vector in an interval of time, instruction count, or other suitable metric (time periods, processor frequency or cycles, and the like). The decay parameter and rate can be programmable. The rate and/or period can vary with the sensitivity meter, also possibly in a programmable manner, based on conditions such as type of network (home, public, work), activity (gaming, web browsing, office or scientific applications), and other conditions, for example multiple taints from a known particularly untrustworthy source. The rate and/or period can also vary according to hardware environment or perspective, for example whether the hardware is constrained to a fixed rate or enabled for a programmable rate such as via a register loaded by software with pertinent information.

A special instruction can be created to facilitate setting of the sensitivity meter. The instruction can operate in conjunction with the operating system to read a register indi-

cating the level of protection and can change the rate in response to operation of the sensitivity meter.

A Taint Adjustment Vector (TAV) can be formed to adjust rate and period dynamically. The TAV can comprise a timer register which can automatically decrement a set of rates. In an example of TAV operation, the TAV including one or more taint adjustment vector parameters can be applied to the Taint Vector (TV) upon expiration of the timer. In various implementations, the TAV can be applied to the TV by adding the TAV to TV, adding a delta, adding another selected value, shifting, shift-add, multiply, divide. Multiple timers can be used to enable decay for one type of information to be different from decay for another type of information. Taint Adjustment Vectors or timers can be universal over multiple Taint Vectors or per Taint Vector.

A special instruction, for example a system-level "set taint vector parameter" instruction, can be created to support the TAV. The instruction can act under operating system control in conjunction with multiple timers, each of which controls a set of taint adjustment parameter vectors (TAVs) which are used to adjust the current taint vector. The instruction can set the TAV and/or timer. The instruction can write to a control register and allocate the control register in control register space as a TAV or timer.

Another technique for delay can be recursive addition of a Taint Bias Vector (TBV) to the Taint Vector (TV), enabling the operating system to create complicated algorithms in the operating system time stamp independently of hardware operation and thus enabling flexibility in modifying, selecting, and executing the algorithms. The algorithms can generally include primitive operations such as a shift, an add, and a subtract, although any suitable operation can be performed. TBV can be larger in number of bits than TV. Bias can constrain software functionality, for example increasing or decreasing the level of sensitivity based on relatively complicated factors since the software may not be completely trusted. Bias can also constrain operation by preventing instant decay (bias may not be allowed to fully eliminate security), although the operating system can be configured to authorize or enable setting of instant decay.

In various system embodiments, taint vectors can be configured with selected taint elements to describe selected taint events.

Accidental/non-malicious overflows can be taint events. Taint handling can be constituted to handle legitimate overflows which can occur sporadically and can be expected to occur. Overflows are examples of known problems. Special instructions can be created to address such known problems. Hints can be used in association with instructions, for example by hint instructions which are dedicated to hint handling or by adding a hint bit field to an instruction. In the case of overflow, a hint can be used to notify that a particular instruction, for example the next instruction, may overflow.

Hint handling can be added to a taint vector, or to an "ignore problems" variety of taint vector. For example, a HINT instruction can be constituted that, rather than the occurrence of a taint causing accumulation of the taint vector, a count can be added to an Ignore Problems Taint Vector (IPTV).

A predictive hint can also be used to allocate resources. For example, a software routine can use a hint a prediction of a significant amount of floating point usage. A HINT instruction can be included in the routine. In another version, at the beginning of a library function, code can be inserted to enable predictive preferential scheduling. The HINT instruction can be part of the library, for example at the beginning, or associated with the library. Code can be

13

inserted in the library, such as at the beginning of a library function requesting particular resources, for example for preferential scheduling. In one example form, a call to a system call can request the operating system to allocate more resources. In another example form, a hint instruction can be sent to hardware to implement the hint and the hardware responds by using the hint in hardware scheduling, such as push, pop, pull, stack, or the like. The hint instruction typically has no direct effect on program execution. The program will run correctly except for changes in performance and battery life.

Predictive hints can also be implemented other than with a hint instruction. Rather than an instruction, the hint may be part of the data structure. For example, X number of bits can relate to expected capabilities to which a process can be entitled such as a vector or a structure. Software can determine information for a performance descriptor, then fills in the data so that metadata of a descriptor determines importance of the performance descriptor.

Accordingly, predictive hints can be implemented in hardware, software, the instruction set architecture, or a combination of configurations. Hardware is typically more constrained than a software implementation. A software library enables the hint to be passed in a linked list of hash trees for passage into hardware, for example as a 128-bit or 256-bit register. Such an implementation can be implemented in an application programming interface (API) but sufficiently simple to be part of hardware. Thus, the API can be designed, then simplified sufficiently to put into hardware.

A taint vector can be used to simultaneously manage, monitor, analyze, and respond to taints associated with various resources independently from one another. The taint vector can be specific to particular sources and resources including networks, systems, processors, memory, hardware, software systems, virtual entities, and the like, including various aspects of operation. The taint vectors can operate on a resource pool and enable detection and resolution of various types of taints. Accordingly, the taint vectors can be used to manage security risks and faux pas of resources.

In computing systems, operating systems, network systems, and the like, a trap, which can also be termed an exception or fault, is generally a type of synchronous interrupt which can be caused by an unusual or exceptional condition. Examples of such conditions can be an overflow, breakpoint, invalid memory access, division by zero, and the like. A trap can result in a switch to kernel mode in some computing systems so that the operating system performs a predetermined action before returning control to the originating process. A trap in a system process is generally more serious than a trap in a user process. A trap can be an interrupt intended to initiate a context switch to a monitor program or debugger.

A trap can be thrown based at least partly on a current value of an entry of a taint vector. As taint vector entries are increased, a trap can be thrown based on a current value of a taint vector entry or one or more thresholds. Thresholds can be varied by affiliate, current system characteristics or activities, or the like. Conditions can involve one or more comparisons between taint vector entries and thresholds. Thresholds can be entry-specific, apply to similar affiliates, be universal, be applied for a sum of taint vector entries, and the like. Thresholds can be set or changed to reflect different tolerance levels. A variety of target functions, weights, masks, etc. may be implemented. Referring to FIGS. 1A, 1B, 1C, 1D, and 1E respectively, first, second, third, fourth,

14

and fifth schematic block diagrams depict embodiments of a computing system 100 that is operable to conditionally trapping based on a taint vector 104. FIG. 1F is a graphical data description showing an aspect of operation of the computing system 100. FIGS. 1G and 1H are data structure diagrams illustrating example embodiments of taint vector 104. A computing system 100 can comprise at least one taint vector 104 operable to list at least one of a plurality of taints 105 indicative of potential security risk originating from at least one of a plurality of resources 102, and response logic 106 operable to monitor the at least one taint vector 104 and respond to a predetermined taint condition 108.

As shown in FIG. 1G, trapping can be based on a taint vector 104 to facilitate intrusion detection using a taint mechanism. In various embodiments, the computing system 100 can be configured such that the plurality of taints 105 comprise one or more of a plurality of distinct classes 110 comprising a plurality of distinct sources 112, events 114, activities 116, and/or conditions 118.

The computing system 100 can be constituted wherein ones of the at least one taint vector 104 comprise a plurality of entries 120 selectively allocated to ones of the plurality of taints 105.

Referring to FIG. 1A, in various embodiments and/or conditions, the computing system 100 can also respond to a detected security risk event and/or condition. For example, the computing system 100 can be configured wherein the response logic 106 is operable to trap based at least partly on a current value of an entry 120 of the at least one taint vector 104.

The computing system 100 can respond to the detected security risk event and/or condition in a predetermined manner. For example, various embodiments of the computing system 100 can be operable to respond to security risk upon determination of the at least one security risk event by a response selected from various responses such as ignoring a security risk event, logging the at least one security risk event, displaying a notification, displaying a warning message, generating an alarm, and the like. Other responses can extend beyond passing of information to dynamic management and control of system operations such as preventing a memory and/or register write, modifying operating frequency, modifying operating voltage, modifying an operating parameter, performing a system call, and the like. Even more drastic responses can terminate a particular process, and end operations of some or all resources, for example by calling a trap and/or exception, terminating operation of selected resources, activating a system shutdown, and the like.

In various embodiments, the computing system 100 can implement one or more "Trap-If Rules" that enables trapping, for example to a predetermined software process, for any selected condition. Typically, the rules can be simple and easy to describe and implement. Accordingly, the response logic 106 can be operable to accumulate taints 105 in at least one entry 120 of the at least one taint vector 104 to trap based at least partly on a current value of an accumulated entry 120 of the at least one taint vector 104.

In some embodiments and/or applications, the computing system 100 can be constituted such that the response logic 106 is operable accumulate taints 105 in at least one entry 120 of the at least one taint vector 104 to trap based at least partly on a comparison of a current value of an accumulated entry 120 of the at least one taint vector 104 to a threshold.

The taint vector 104 can be an accumulator of faux pas such as a null pointer reference, an attempt to access a secure part of the CPU, a buffer overrun or other similar common

15

hacking techniques, and the like. A taint can be any event or condition that may be innocent or coincidental but possibly originating in a region that raises suspicion. Various regions or sources can be assigned various levels of suspicion. Thus, the computing system 100 can implement various target functions for analysis of taints 105 to attain optimization in determining security risks. In some implementations and/or conditions analysis can focus more on the type of malicious event that occurred rather than the origin. Hence, the computing system 100 can be formed such that the response logic 106 is operable to apply at least one function 122 to the at least one entry 120 of the at least one taint vector 104.

In various embodiments and/or applications, the computing system 100 can be constituted wherein the at least one function 122 is selected from a group consisting of weights, masks, sums, combinations, arithmetic functions, logical operations, and transforms.

In various embodiments, the computing system 100 can be configured wherein the response logic 106 is operable to respond to at least one security risk event with at least one response selected from responses including ignoring the at least one security risk event; logging the at least one security risk event; displaying a notification; displaying a warning message; generating an alarm; preventing a memory and/or register write; modifying operating frequency; modifying operating voltage; modifying an operating parameter; performing a system call; calling a trap and/or exception; terminating operation of selected resources 102; and activating a system shutdown, and the like.

In various embodiments, the computing system 100 can handle taints 105 of various types. For example, the computing system 100 can be configured such that one or more of the plurality of taints 105 can be selected from among null pointer references, attempts to access a secured part of a processor, attempts to access a secured resource, buffer overruns, events originating in a region that raises suspicion, faults, integer overflow, multiple taint indicators that exceed at least one predetermined threshold, a taint indicated by power law analysis, a taint indicated by a race function, attempts to access a key, and many others.

The taint vector can be implemented to include tolerances or thresholds set based on questionability of the source or event. Zero tolerance can be set for particularly suspicious or harmful events and/or sources wherein a single event can result in a maximum response. For a low threshold, the response for one taint can result in a trap, exception, or shutdown, and may be used, for example, in nuclear power plant control. A medium threshold can be a hybrid of low and high threshold and call for a medium response and include aspects of decay. An example of high threshold can allow twelve taint counts per unit time such as for cheap video forwarded from a provider or signals from ubiquitous cell phones. Most events can be ignored in the absence of some indication of attack. Thresholds are set to balance a sufficient level of security with communications characterized by large amounts of data and frequent errors. If taints exceed the threshold, then suspicion is sufficiently great that some action or response is taken. A suitable response can be trap, exception, notification, alarms, and the like. Accordingly, referring to FIG. 1B, the computing system 100 can further comprise thresholding logic 124 operationally coupled to the response logic 106 which is operable to set a threshold for application to at least one entry 120 of the at least one taint vector 104. The response logic 106 can be operable to trap based at least partly on a comparison of the at least one entry 120 of the at least one taint vector 104 to the threshold.

16

In some embodiments, the computing system 100 can be formed wherein the thresholding logic 124 is operable to set or modify a threshold for application to at least one entry 120 of the at least one taint vector 104 based on a tolerance level selected on the basis of consequences of a security risk event.

Thresholds can vary by affiliate, current system characteristics or activities, conditions, and the like. Conditions can involve one or more comparisons between taint vector entries and thresholds. Hence, the computing system 100 can be configured wherein the thresholding logic 124 is operable to set a threshold for application to at least one entry 120 of the at least one taint vector 104 mutually distinctively for a plurality of affiliates 126, system characteristics 128, sources 112, events 114, activities 116, and/or conditions 118.

In some embodiments, the computing system 100 can be formed such that the thresholding logic 124 is operable to set a threshold for application to at least one entry 120 of the at least one taint vector 104 specifically to the at least one entry 120.

Thresholds can be selected as entry-specific, applied to similar affiliates, applied universally, applied as a sum of taint vector entries, and the like. Accordingly, the computing system 100 can be configured wherein the thresholding logic 124 is operable to set a threshold for application to at least one entry 120 of the at least one taint vector 104 uniformly for application to similar affiliates 126.

Similarly, in some embodiments and/or applications the computing system 100 can be formed such that the thresholding logic 124 is operable to set a threshold for application to at least one entry 120 of the at least one taint vector 104 universally wherein all entries 120 are compared to the threshold for a plurality of affiliates 126, system characteristics 128, sources 112, events 114, activities 116, and/or conditions 118.

The computing system 100 can thus be formed to trap if the taint vector 104 meets predetermined condition(s), such as of the form  $v_i > Th_i$ , where  $v_i$  is an element in the taint vector 104 and  $Th_i$  is the threshold for a taint  $i$ . The threshold  $Th_i$  can be applied for each element of vector, for example any elements ( $v_i > Th_i$ ), all elements ( $v_i > Th_i$ ), the sum of elements, for masked elements, if any  $v_i$  is greater than a global threshold, and the like. For example, the computing system 100 can be formed such that the thresholding logic 124 is operable to set a threshold for application to a sum of entries 120 of the at least one taint vector 104.

In some embodiments, the computing system 100 can be formed with thresholding logic 124 operable to set or modify a threshold for application to at least one entry 120 of the at least one taint vector 104 based on variation of tolerance level by application of a predetermined weight function.

Furthermore, the computing system 100 can be constituted wherein the response logic 106 is operable to respond to at least one comparison between the at least one entry 120 of the at least one taint vector 104 and a plurality of thresholds.

Taints can be generated on the basis of questionability of the data and of other aspects of operation and condition such as prior negative experience or lack of familiarity with a data source or entity. Referring to FIG. 1C, embodiments of the computing system 100 can further comprise monitoring logic 130 which is operationally coupled to the response logic 106 and operable to accumulate the plurality of taints 105 arranged as at least one entry 120 of the at least one taint vector 104.



17

In various embodiments, the computing system **100** can be constituted to implement a wide range of accumulation functions. For example, the computing system **100** can be formed such that one or more of the at least one taint vector **104** are operable as an accumulator of a plurality of taint indicators indicative of potential security risk from a plurality of distinct sources **112** at distinct times. One or more of a corresponding plurality of distinct accumulation functions can include comparing ones of the accumulated plurality of taint indicators to at least one predetermined threshold, performing power law analysis, and/or performing a race function. The computing system **100** can be formed such that one or more of the at least one taint vector **104** are operable as an accumulator for counting the number of tainted instructions. Other suitable accumulation functions can operate by counting various occurrences or aspects of operation such as counting the number of taints **105**, counting the number of taints **105** per memory unit, counting the number of instructions tainted, counting the number of instructions written as a result of a taint, counting the number of data loads and stores, counting the number of memory accesses, counting the number of calls, counting the number of returns, and counting the number of branches. Still other counting aspects of accumulation functions can include counting the number of integer overflows, counting the number of network input/output events, counting the number of null pointer references, counting the number of buffer overruns/overflows, counting the number of repeated attempts to access a key, and the like. Suitable accumulation functions can be used to monitor any aspect of operation.

Referring to FIG. 1C in combination with FIG. 1H, The monitoring logic **130** can be operable to determine whether information from a particular source or entity is trusted based on assessment of security risk. Thus, in a further aspect of operation, the computing system **100** can further comprise monitoring logic **130** operationally coupled to the response logic **106** which is operable to acquire and monitor a history of the ones of the at least one taint vector **104** in a feedback loop **142** that correlates taints **105** with responses to the taints **105**.

In various embodiments, the computing system **100** can be operable to specify at least one of a plurality of decay options selected from applying decay after a predetermined number of operations to avoid triggering on outlying events, setting decay to account for rare and spurious events with a probability of occurrence by chance during long term monitoring, incrementing/decrementing using a single vector, and/or subtracting a predetermined number. Other suitable decay options can include shifting a taint vector **104** in an interval of time, shifting a taint vector **104** at a predetermined instruction count, shifting a taint vector **104** at a predetermined processor cycle count, copying a taint vector **104** periodically to memory to maintain an old version while incrementing/decrementing to enable restoration following an invalid or error condition, and imposing decay that balances accumulation. Further examples of suitable decay options can include applying decay periodically, applying decay with a varying period that varies based on a sensitivity meter, applying decay with a varying period that varies based on environment, applying decay with a varying period that varies based on activity type, applying decay according to a programmable parameter at a programmable rate, and the like.

In various embodiments, the monitoring logic **130** can be constituted to perform various tracking and monitoring operations to enable enhanced detection of intrusion. For example, in some embodiments the computing system **100**

18

can be configured wherein the monitoring logic **130** is further operable to track taint indicators characterized by a range of taintedness from potentially suspicious to definite taints **105**.

The monitoring logic **130** can be constructed to perform various comparisons to indicate error or intrusion. For example, the computing system **100** can be configured wherein the monitoring logic **130** is further operable to monitor comparisons selected from a group including determining whether any elements are greater than a predetermined threshold, determining whether all elements are greater than a predetermined threshold, determining whether the sum of some elements is greater than a predetermined threshold, determining whether the sum of all elements is greater than a predetermined threshold, and the like.

The computing system **100** can be configured to discern actual security risks from innocent and/or coincidental events. For example, in some embodiments the monitoring logic **130** can be operable to monitor the plurality of sources **112**, events **114**, activities **116**, and/or conditions **118** to detect and discern one or more potentially innocent and/or coincidental events such as null pointer references, attempts to secure part of a processor, innocent and/or coincidental events arising from a region that raises suspicion, and the like.

In some embodiments, the computing system **100** can further comprise monitoring logic **130** operationally coupled to the response logic **106** which is operable to update the taint vector **104**, process the taint vector **104**, and determine a security risk condition based on the processing of the taint vector **104**. The response logic **106** can be operable to trap in response to the security risk condition.

Referring to FIG. 1D, a computing system **100** can be operable as at least part of a federated system which can be implemented in an infrastructure such as an architecture including servers and clients. For example, gaming code servers and gaming console clients can interact by running program code that executes in part on machines controlled by the server and in part on machines controlled by the client. Intrusion detection via accumulation of taints can enable the interaction to be mutually trusted by both sides. In an illustrative embodiment, the computing system **100** can be operable as at least part of a federated system comprising a least a first source **112(1)** and a second source **112(2)**. Accordingly, the computing system **100** can be configured such that one or more of the at least one taint vector **104** can comprise an entry **120** that is allocated to selected one or more of the plurality of resources **102** wherein taints **105** of the selected one or more of the plurality of resources **102** are federated to the entry **120**. A taint indicator of the first source **112(1)** can be tracked against at least one of the taint indicators of the second source **112(2)**.

The computing system **100** can be operable as at least part of a networked system including multiple computing devices such as computing system **100** which interfaces with remote and potentially untrusted computers and may be the source of security risk events such as attacks. Security risk events or attacks can arise from other sources including computing devices and systems, storage and devices inside a firewall or local to a targeted machine. In general, computers and networks can represent a variety of local or globally distributed systems and networks that can supply information via a plethora of communication channels and protocols such as the Internet. Thus, the computing system **100** can further comprise monitoring logic **130** operationally coupled to the response logic **106** which is operable to

monitor taints **105** in network **134** input/output operations. The response logic **106** can be operable to trap to a software process **132** based at least partly on determination of a network **134** input/output condition of an attempt of malware **136** to communicate to a malware operator **138**.

Security risk events and attacks can originate remote from a local and potentially trusted network, and can similarly originate from local users, systems, devices, and storage. Accordingly, the computing system **100** can further comprise monitoring logic **130** operationally coupled to the response logic **106** which is operable to monitor taints **105** using a hardware device **140**. The response logic **106** can be operable to insert initial taint notifications using a software process **132**. In some embodiments, the computing system **100** can be constituted to address security risk events that arise from a local device such as keyboard, network interface, communication devices, local storage including memory and long-term storage devices, and other computers and systems.

The systems and techniques disclosed herein are operable in the context of physical hardware and software-oriented configurations. The systems and techniques are further operable for embodiment as virtual computers and devices presented or emulated within a virtualization system. Thus, the computing system **100** can be used in physical hardware systems, virtualized systems, and combination systems with both physical and virtual aspects, with functionality distributed across devices or systems. Thus, taint information can be received from a source remote from a targeted system, such as from an interface, a network, a gateway, remote computer, or the like.

Taint information can be received from some source and can be destined for some target storage location and downstream usage. Information or data can be considered tainted, potentially tainted, suspect, or known untainted based on multiple criteria. Tainted information or events are defined according to a particular implementation and security policy in a range from “of interest,” potentially untrusted, and suspect to untrusted, potentially dangerous, and malicious. Information can be considered tainted based on entity including source, target, and interface; and also based on characteristics or conditions of information receipt such as conveying protocol or transaction; or based on a combination of considerations.

In some embodiments, shown in FIG. 1G, the computing system **100** can be configured wherein one or more of the at least one taint vector **104** comprise a composite taint vector **146** that correlates a taint source **148** and a taint activity type **150**.

Referring to FIGS. 1D and 1G, in various embodiments and/or applications of the computing system **100**, taints **105** can be integrated with a processor **152**. Accordingly, the computing system **100** can further comprise at least one processor **152** included in the plurality of resources **102**. The at least one taint vector **104** can be selectively positioned within the at least one processor **152** proximal to at least one source at which a taint originates.

In some embodiments, the computing system **100** can further comprise at least one processor **152** included in the plurality of resources **102** in a configuration wherein the at least one taint vector **104** comprises a single taint bit corresponding to the processor **152** to indicate a taint indicative of potential security risk associated with the processor **152**. In some embodiments and/or applications, the processor **152** can have at least one register integrated

into the processor **152**. A taint vector **104** can indicate a taint indicative of potential security risk associated with the register.

Taints can be generated in association with operation of a translation lookaside buffer (TLB) **154** in the processor **152**. A translation lookaside buffer (TLB) **154** is a processor cache which can be used by memory management hardware to improve virtual address translation speed. Processors use a TLB to map virtual and physical address spaces. TLB are used widely in hardware which uses virtual memory.

The TLB **154** can be implemented as content-addressable memory (CAM), using a CAM search key which is the virtual address to produce a search result which is a physical address. If the TLB holds the requested address—called a TLB hit, the CAM search quickly yields a match and the retrieved physical address can be used to access memory. If the TLB does not hold the requested address—a TLB miss, the translation proceeds by looking up the page table in a process called a page walk. The page walk is computationally expensive process, involving reading contents of multiple memory locations and using the contents to compute the physical address. After the page walk determines the physical address, the virtual address to physical address mapping is entered into the TLB.

A stream monitoring instruction can be implemented to improve efficiency and performance of the TLB by supporting a software predictor. The instruction can be used to monitor misaligned or split access. A memory access is aligned when the data item accessed is n-bytes long and the data item address is n-byte aligned. Otherwise, the memory access is misaligned. Monitoring for misaligned access can be performed by hardware, resulting in a trap, or somewhat less efficiently by software. In practice, monitoring for misaligned access has a high false positive rate, for example approaching 90%. A predictor can be configured, for example by microarchitecture adjustment or taint accumulation, to indicate whether the misaligned access hits are accurate.

The processor **152** can be configured to change voltage, frequency, and/or power based on the number of cache misses. For example, logic can accumulate taint indicators to detect an abundance of cache misses or other performance problems, the voltage can be varied such as increased to cure the problem. The logic can dynamically adjust operating parameters according to the amount of traffic. Frequency and voltage can be adjusted, for example whenever a change in frequency occurs, the voltage can be modified accordingly.

Logic in a memory interface can detect when memory is full to some threshold level, for example 70%, for example by accumulating taint indicators. If memory is full to the threshold level, a predetermined taint indicator condition is found, and a high level of access is occurring, memory speed can decrease. In response, the frequency and voltage of operation can be dynamically increased to maintain a desired memory speed.

In various embodiments, logic for performing dynamic adjustment can be positioned in memory, in a logic interface, in a processor. A hardware configuration can optimize by active adjustment, redirection, or possibly a combination of adjustment and redirection. For example, a computation-intensive process with many instructions to be executed rapidly can be addressed by running the processor at a higher rate by increasing operating frequency and voltage, and/or some of the burden can be shifted to components other than the processor to maintain processor execution at a lower frequency.

Taint accumulation can also be used to allocate system resources. Various aspects of resource allocation include hardware threading, computational limits, pooled resources, entitlements, and others. Resource allocation can be handled via various architectural aspects of a system including microarchitecture, instruction set architecture (ISA), operating system, library calls, and taint accumulation. Software can associate capabilities with particular library functions or software objects. This software can be in the form of compiler, operating system, or others. The operating system can, for example, create a profile for any process running floating point operations and give that entitlement. Resources allocated include processors, central processing units (CPUs), graphics hardware, network controllers, memory, memory management, other hardware, and the like. Resources further include power, cycles, and the like.

#### Hardware Threading.

Several aspects of hardware threading are currently implemented in processors such as CPUs. Simultaneous threading (SMT), hyperthreading, or simultaneous hyperthreading relate to hardware execution of two or four threads selected for running at any time, managed according to many fine-grained scheduling decisions. In a cycle, two threads are selected at instruction fetch, typically at the front of the pipeline and hardware determines which of the two threads' instructions to fetch. An instruction for each of the threads pass to an out-of-order machine within which the instructions are running concurrently. For example, an arithmetic logic unit (ALU) instruction from thread 1 and a memory instruction from thread 2 can run simultaneously.

Another type of hardware threading is interleaved multithreading (IMT) which removes all data dependency stalls from the execution pipeline. One thread is relatively independent from other threads so the probability of one instruction in one pipeline stage needing an output from an older instruction in the pipeline is low. IMT is conceptually similar to pre-emptive multi-tasking used in operating systems.

In contrast to CPU multithreading which handle relatively few threads (typically two or four threads), graphics processing units (GPUs) are stream processors for computer graphics hardware and manage hundreds or thousands of threads, thus using much more sophisticated scheduling. When blocking occurs, for example on a cache miss such as from a memory reference, a very large number of threads are blocked. Threads are chosen for execution on massively parallel thread arrays. In a typical arrangement, a processor has approximately 64,000 threads of which only about a thousand execute at one time. Underlying operations during execution include scheduling, addressing cache misses, and the like. Rather than scheduling from a memory pool, GPUs schedule instructions for execution from a very large pool of threads, waiting for memory to become available to run the next thread.

A CPU can be configured for a CPU thread hierarchy which includes a currently running list and a pool of non-running threads enabled to receive information pertinent to computational limits from devices or components such as special-purpose hardware. In an illustrative embodiment, the information pertinent to computational limits can be monitored via taint indication and taint accumulation, and resources allocated accordingly.

Computational limits can be imposed via generation of taint indicators and taint accumulation. A limit on computation can be imposed according to setting of priority level which is, in turn, based on available resources. One example resource that can be monitored to set limits on computation

is the battery. Limits on computation can be imposed based on battery consumption, battery life remaining. Computational limits can be addressed via a framework of setting capabilities, for example specifying a capability to execute on selected processing resources. In an example implementation, the capability can be set up in metadata.

Taint accumulation is suitable for managing computational limits since addressing computational limits can be fairly complex, involving not only information from monitored resources but also user input. For example, a determination by hardware of low battery level and associated limited battery life can be overridden by a user who may request a software application to run in anticipation of being able to soon recharge the battery at a line power source.

Performance capabilities can be used in combination with taint accumulation to manage resources. A performance capabilities framework can be defined to address handling of a pool of available resources. A thread pool pattern can be configured wherein a number of threads are created to perform a number of tasks which are typically organized in a queue. Usually, the number of tasks is greater than the number of threads. A thread upon completing an associated task will request the next task from the queue until all tasks have completed. The thread can then terminate or become inactive until new tasks are available. The number of threads can be tuned to improve performance, and can be dynamically updated based on the number of waiting tasks. Increasing the size of the thread pool can result in higher resource usage.

A hardware scheduler can respond to any countable or measurable operating condition or parameter, for example electrons, constraints, frequency, cycles, power, voltage, and the like, to control the thread pool and pool of resources. The countable or measurable operating conditions and/or parameters can be monitored over time using taint accumulation. Two highly useful conditions or parameters for monitoring are power and cycles, which are the basis for other measurable phenomena. Monitoring of operating conditions can be performed in hardware or via software call.

Furthermore, software can associate capabilities with particular objects such as libraries.

Taints for main memory may be located in different places, organized differently among different memory locations or types, and/or associated with different amounts of memory. In an example configuration, a single taint vector can be allocated for all main memory, which can have multiple entries that are associated with different memory ranges. Taints can be organized by or applied to memory pages. Level of taint may be indicated per memory block. A separate taint table can be used if a general page table is read-only, and a size of a separate taint table may be reduced via hashing. Taints can be applied on a per-byte basis, but then likely only 1-2 bits and significant overhead. Accordingly, as shown in FIG. 1E, taints can be associated with main memory to facilitate intrusion detection in a taint mechanism. Thus, various aspects of taint vector location and/or granularity can be formed in memory.

In some embodiments, the computing system **100** can further comprise at least one memory **156** included in the plurality of resources **102**. The at least one taint vector **104** can be selectively positioned in a respective at least one location within the memory **156**. In some implementations, taints **105** can be selectively distributed in a plurality of locations and organized distinctively for one or more of the plurality of taints **105**. Multiple taints **105** can be associated with selected amounts of memory **156** individually among

one or more of the plurality of taints 105. If desired, a single taint vector 104 can be allocated for an entire memory 156.

In some embodiments, a single taint vector 104 can be allocated for an entire memory 156 and comprise a plurality of entries 120 associated with different ranges in the entire memory 156. Similarly, one or more taints 105 can be applied to the memory 156 on a per-byte basis. Granularity can be managed on the basis of a memory page. Similarly, one or more of the plurality of taints 105 can be applied to respective memory pages.

In various embodiments and/or applications, granularity can be managed on the basis of a memory block, for example with a level of taint indicated per memory block. The memory 156 can include memory blocks with different memory types. Granularity of the memory types can be selected mutually distinctively such that the size of a memory block per taint vector 104 is mutually distinctive. For example, the memory 156 can include volatile and nonvolatile memory. More specifically, a read-only page table and a read-write taint table can be allocated such that the size of the read-write taint table can be reduced by hashing. Thus, in some embodiments, the computing system 100 can be configured for handling a memory taint hash wherein logic can be operable to form a memory taint hash table and use the memory taint hash table to indicate a level of taint per memory block. Thus, the memory taint hash table can be accessed to indicate a level of taint per memory block using the memory taint hash table.

In an example software embodiment, software can monitor the system over history, or can be preprogrammed, and fills in some sets in entitlement vector fields. Software can determine values for the fields and fill in the bits of data, possibly associated as a lookup table, an associated hash table, an extra field to call for a library, and the like. For a library call, an entitlement vector EV is returned. The entitlement vector can be received from various sources, for example from external to calling software. For example, the entitlement vector EV may be installed into hardware as a side effect of the library call.

A factor in determining whether the entitlement vector is handled in software or hardware is the size of the vector.

In an example hardware implementation, a suitable entitlement vector size is 256 bits, although any suitable size is possible. For example, a vector of 64K bits is generally considered too large for hardware implementation.

In some embodiments, an entitlement vector can be associated with each library. The entitlement vector can be used, for example, to eliminate floating point if desired, reduce the number of floating point operations if such operations are rarely used, reduce the scale as appropriate when full accumulator width is unnecessary, increase support for the ALU.

The entitlement vector can be implemented as a call with a memory address made in association with a call to a library which, for example, can return a pointer or address location to the entitlement vector.

Another field of the entitlement vector can be a chooser/thread selector. The entitlement vector can be used by the chooser/scheduler, which includes logic that performs operations based on a single entitlement vector or possibly relative entitlement vectors. Each Instruction Pointer (IP) or thread can have an associated entitlement vector. For example instruction pointers, for IP1, IP2, IP3, IP4, then four entitlement vectors can be allocated. Chooser/scheduler logic considers the entitlement vector when scheduling the next thread for computation. The logic informs the chooser/scheduler about how to make the selection. The logic can

perform selected functions to make the choice and for scheduling, for example by elevating or decreasing priority of a thread.

An example function using an entitlement vector (EV) can compute the sum of weight times EV; compared to the usage vector of Thread, a simple target function for evaluating when to schedule threads from the highest priority to the lowest priority. Thus, for a thread with high priority and large requirement for resources, the thread can be elevated in the scheduling list and resources are likely to be allocated. In contrast, a thread that is a glutton for resources and has low priority is likely to be deferred by the scheduler, moving back or to the end of the list of scheduled threads. A high priority thread that consumes only limited resources is likely to be moved up in the schedule list, possibly to the front of the list.

In some embodiments, the entitlement vector supplied by a HINT instruction can be modified by a capability process. Illustratively, the entitlement vector can set entitlement to use X resources which can be limited by the operating system for example by reduced weighting or setting of maximum allowed resources. The entitlement vector can also be limited according to usage, wherein a thread using an inordinately large amount of resources can be limited when the high usage is detected or predicted.

The entitlement vector function  $F_i(w_i, EV_i, v_i)$  of weight ( $w_i$ ), entitlement vector ( $EV_i$ ), and resource volume ( $v_i$ ) can be either linear or non-linear.

The entitlement vector enables association of scheduling with functions. The entitlement vector further enables association of priority with functions.

One of the challenges in allocating resources is the potential for highly unpredictable changes in resource demand. For example, minor changes in workload can result in substantial variation in performance. Another challenge is unpredictable behavior in response to context switches from one process to another. One technique for dealing with these challenges is making a library call as a technique for determining whether a context switch occurred or, if not expecting to make a library call, perform an action that randomizes priority. If degradation results from making the library call, then performance can be monitored to determine whether performance is reduced. If so, priority of the threads can be randomized. Example techniques for randomization can include a Boltzmann search, simulated annealing, hop-around, other lateral computing techniques, and the like. A Boltzmann search can be performed by a Boltzmann machine, a stochastic recurrent neural network that is capable of learning internal representations and solving combinatoric problems. Simulated annealing is a computer technique used for answering difficult and complex problems based on simulation of how pure crystals form from a heated gaseous state. Instead of minimizing the energy of a block of metal or maximizing strength, the program can minimize or maximize an objective relevant to the problem at hand, specifically randomization to attain stable performance. In a hop-around technique, priority or other parameters can be bounced around to determine a local maximum but not global optimum. Search optimizations can be used to determine whether truly at a maximum value. The new results can be compared with an old optimum.

In some embodiments, a supervisor circuit, for example for thermal and/or overvoltage protection, can modify the entitlement vector.

The entitlement vector, for example in combination with a usage vector and/or taint accumulation monitoring, can be used for monitoring power control. In various embodiments,

25

power control monitoring can be performed remotely or locally, possibly by the operating system.

In an example embodiment, a user can supply an entitlement vector using instructions, for example by specification of the beginning and end of a function. The entitlement vector can be used in association with a performance monitoring unit which monitors and determines other entitlement vectors. In various embodiments, the entitlement vectors can be maintained separately or combined into a single effective entitlement vector.

Context switches can be specified as taint indications for usage in taint accumulation. Context switches can be defined as switches from one process to another. In contrast, a thread can typically be considered limited to a single context. Standard threads and mock threads share resources including context and can have multiple processes, multiple threads within the same privilege level technically. However, a threading library and threading operating system can be created wherein threads are not limited to the same context. Threads can comprise simply a stack and an instruction pointer, and can run in the same address space, for example threads can run as different users in the same address space. In a case of multiple users accessing the same database, if the database is a shared-memory database, software or an interpreter can be responsible for ensuring that unauthorized user(s) cannot access certain data. In the case of users assigned different privilege levels or different threads in the same virtual memory address space assigned different privilege levels, different registers are assigned to particular users and/or threads, and thus switches between users and/or threads are context switches.

Privileges can be associated with a page, a page table, an actual physical memory address, a virtual memory address, and the like.

Capabilities and entitlement can be used in combination with taint accumulation for managing resources. In some embodiments, the capabilities vector and the entitlement vector can be merged. In some aspects of operation, entitlement can be considered to be a capability. With entitlements specified, the associated performance capabilities and management of associated capabilities prevents unauthorized access to data and/or resources, and prevents system takeover, unless specifically allowed or enabled by a system call, improving security and enabling denial of service to attacks.

Merged capabilities and entitlement can be used to prevent microarchitectural denial of service. Denial of service is typically considered to arise from a hacker on a network blocking access by using up all or a substantial part of network bandwidth. For example, when operating on a virtual machine in a cloud computing platform (such as Amazon Elastic Compute Cloud (EC2)) a job can be run that thrashes the cache, resulting in an architectural denial of service in response. Preventative remedies can include checking for performance counters and preventing such unauthorized accesses. Microarchitectural remedies can also be used such as implementing microarchitectural covert channels in which, for various types of code, secret keys running on the same virtual machine can be detected. Similarly, microarchitectural covert channels can be used to monitor timing of code to detect intrusion and to detect whether a bit is set in a particular bit position which may indicate intrusion. Microarchitectural techniques can thus include timing channels and covert channels for use whenever a shared resource is to be modulated. Covert channels can be applied, for example, in modulating a disk arm, detecting seeks on a file system.

26

In various embodiments, operations implementing and using the entitlement vector can be executed by software in a processor, by microcode, in logic, in hardware, or the like.

An infrastructure configured to support multiple processors in a system can have a shared memory and message passing between threads, processes, processors, and the like. Operating systems (OS) can include various mechanisms to enable message passing, for example pipelines, daemons that use sockets, loopback, and the like. Any suitable number of processors can be supported in the system, from relatively small systems with few processors to large scale systems with hundreds of thousands or millions of processors. In a typical large scale system, the multitudes of processors communicate via fat trees which support the large amount of bandwidth demanded by the large scale system. The amount of bandwidth in different positions in the tree is variable, depending on traffic. In various other configurations, the many processors can communicate via meshes or buses, via Gigabit Ethernet, via CDMA-CE (Code Division Multiple Access-series CE), and the like. In large interconnects, the number of processors determines what functionality is attainable. For example, for more than about 1000 processors, memory can no longer be shared. At around 100 processors, memory space can be shared but cache-coherence is typically not possible and memory is thus non-cache-coherent shared memory. Cache-coherence is generally considered to cause problems for more than about sixteen processors so that fewer processors at a first level can have cache-coherent shared memory.

For a supercomputer or other system with the large number of processors, for example more than about 1000, for which memory is non-shared, Message Passing Interface (MPI) can be used for communication. MPI uses multiple threads but does not use shared memory. The MPI multiple threads are all part of local shared memory, but no global shared memory exists. The amount of local shared memory is limited, resulting in a communications bottleneck. Supercomputer memories use Message Passing Interface (MPI) which, to a first order, includes a limited number of instructions such as send some location, buffer, end buffer, and receive some entity, buffer, end buffer, and the like. MPI is an application programming interface (API) and is thus a library call. The received entity can be, for example, a channel connecting the sender and the receiver, although channels are rarely used in MPI since channels do not scale beyond about a thousand processors. Accordingly, MPI can use commands with masks which identify which processors are to receive a message. A difficulty with MPI is that different code must be written, and a different core engine and interface, for small-scale and large-scale parallelism. Thus, send-and-receive communication such as is used by MPI is suitable if memory is shared.

What is desired is a technique for expanding send-and-receive communication more broadly. In accordance with system and method embodiments, a communications application programming interface (API) can be created that enables communication between different types of threads and hides that the threads are sharing memory. The communications API can enhance functionality of a Transmission Control Protocol (TCP) socket. The TCP socket, also termed an Internet socket for network socket, is an endpoint of a bidirectional inter-process communication flow across and Internet Protocol (IP)-based computer network such as the Internet. In some embodiments, the communications API can also incorporate functionality of MPI into that of a TCP socket. In a distributed system, a processor can communicate with a Network Interface Controller (NIC) and a send

instruction puts data on a queue to send to the NIC and pass through the routing network to a specified destination. The communications API can perform communications via TCP-IP, in some configurations optimizing aspects of TCP-IP such as by ordering packets, and also via other protocols. The communications API can include send-and-receive functionality, and include one or more channels, which is operable with TCP-IP. Some of the channels can be shared memory in the form of a buffer with a counter. Some channels can connect to the NIC, some channels to TCP-IP, and some channels can have other functionality. In some embodiments, the communications API can support different types of channels. One example of a channel type is simply registers. Another type of channel can run two hardware threads with a pipeline coupled between the two threads.

The communications API can be adapted to handle the possibility of overflow. For example, for a channel implemented as shared registers, filling the registers to capacity can cause overflow to memory, which can call a trap or exception. In some embodiments, an overflow condition can be specified as a taint indication and accumulated for resource management.

Another technique for expanding send-and-receive communication more broadly can comprise creating a message passing infrastructure in hardware. Speed is one advantage of forming the message passing infrastructure in hardware. For example in the case of a system call, conventionally a slow operation, hardware can be configured to support a send instruction operable to check a bit in a channel selected for the send operation to determine whether the channel is available and, if not, performing a system call by faulting to the system call. Thus, the hardware can be configured to pass execution through the operating system in response to desired conditions.

In an example embodiment, the message passing infrastructure hardware can be configured to avoid passing execution through the operating system, for example to avoid the context switch inherent with going to the operating system. In another example embodiment, the hardware can be configured to include a message passing paradigm and one core can be run in ring 0 to enable access to operating system calls. The operating system is not a separate process but rather a library call in a library. Another option is to allocate a hardware thread to the operating system.

The operating system performs a ring 0 call via a system call which, in terms of hardware implementation, can be a function call to change a bit, granting permission to change the bit, and identification of the stack from which the OS is operating. In one example implementation, the user can explicitly control the stack, for example by placing the operating system stack in a different register. In another implementation, a system call can change the instruction pointer and the stack.

The message passing infrastructure hardware implementation can, for example, include support for send and receive calls. The hardware implementation can enable faster operating speed. For particular special cases, hardware send and receive calls can be faster than a shared library call. Send and receive are global messages, supporting point-to-point communication in two-party messaging. In some embodiments, the hardware implementation can support put and get APIs to enable sending a message to a designated address asynchronously or synchronously, as selected. The designated address is in a global address space partition, not local load-store. The put and get APIs can handle access to shared physical memory by sending a request to the master or server for the designated memory location. The memory is

hashed across all the global memory space. In the illustrative implementation, get and put can be system calls rather than instructions, thus facilitating global access. Because the get and put system calls are relatively resource-expensive, efficiency can be attained by communicating blocks of data, for example 64K, at one time rather than for individual bytes.

For a cache-coherent shared memory that is accessed using the put and get system calls, different schemes can be used depending on what entities are communicating. For entities which share memory, the get and put calls simply access the shared memory. For entities separated by substantial physical or network distances, the get and put calls, if unable to fulfill the call by shared memory access, by running through the same router or similar local actions can send the calls to the network interface to relay remotely, for example across the world. For shared memory, whether cache-coherent or cache-noncoherent, the put and get, send and receive operations are relatively simple since all entities can access the same memory. More complexity arises when memory is not shared. In various embodiments, when memory is not shared different schemes can be used such as copy-on-write (copying the shared memory), creating in remote memory the shared memory that shares the same capability, an implicit in the put and get, or other options.

The message passing infrastructure thus can include hardware support for the various put and get, send and receive, or the like system calls or instructions. The message passing infrastructure can be configured to enable two threads to be forked and used with the put and get calls to enable optimum speed performance. The send and receive, and put and get instructions, as described, consume two hardware threads or might consume two passive threads.

In some embodiments, the put-get and send-receive can be combined with access bits which designate memory to which the sender is allowed access. Passing along the access bits can enable a reduction in overhead while enabling protection across processes. The overhead of switching or sending a message drops significantly because the receiver already knows the memory to which the sender has access.

In some embodiments and/or applications, taints can be applied to memory segregated by type to facilitate intrusion detection using the taint mechanism. Memory can thus be segregated into different types such as for code and data memory, or the like. Different types can have taint mechanisms applied differently to the different types. For example, different memory types can have different levels of granularities, for example larger or smaller blocks of memory per taint vector. Also, different memory types can have more or fewer bits per taint vector entry. Furthermore, different thresholds, decay rates, and the like can be applied to different entries corresponding to different memory types.

Referring to FIG. 1F, a graphical data description shows an example operation that can be executed by the computing system 100 to facilitate intrusion detection using taint accumulation. In an illustrative embodiment, the monitoring logic 130 operationally coupled to the response logic 106 can be operable to monitor taints 105 and create a trust profile based on the monitoring. The response logic 106 can be operable to trap to a software process 132 based at least partly on determination of a suspicious condition.

In an example embodiment, a software model can be configured to use and enforce performance capabilities. In a relatively simple operation, if power is too low, then the software can limit the maximum number of threads or other capabilities. For example, in a cell processor case the number of threads can be limited to less than 1000. Funda-

mentally, software can disable functionality if sufficient power is unavailable for scheduled operations.

In another example, a sensor or sensors can detect whether battery bias voltage level is recovering too slowly or, similarly, a thermistor can indicate a battery is too hot which may indicate operating at too aggressive a level. A bit or bits can be set indicating the recovery time is too long. The set bit(s) can be used to throttle the maximum thread hopping rate in the case of a CPU with two threads. The bits disallow a thread hop and set an allowable rate of thread hopping; or perhaps allow thread hopping which creates slowing but saves power.

An example of performance capability monitoring and management can be implemented in a CPU with four process threads each having instruction pointers. One of the four threads is selected to execute for next instruction cycle. Various types of information can be monitored to determine which thread to select including recent demand for power, memory, CPU cycles, and the like. For example, a process can be a resource glutton and allocated fewer resources to enable other processes priority. Information is available relating to recent performance, requested performance, and acceptable performance (niceness).

Another option is to use a "NICE" instruction which can be used to adjust the priority level of predetermined instructions, enabling the instructions to be run in the background at a convenient time. For example, if a processor or battery is running too hot, the NICE instruction can reduce the urgency of executing code. In a particular example implementation, the NICE instruction can change a multiplier and step of a decay algorithm.

High and low capabilities can be specified. For example, a particular software routine can sometimes, although rarely, use floating point operations so the capability for such routines can be set low. Operations performed by software can include monitoring, configuring parameters, and the like.

Capabilities can be used to implement security. Typically, a system has only a few predetermined capabilities. However, a system can be configured in which every memory addressing register is assigned a capability. If the register specifies a capability to access the associated memory location, the location can be accessed. Otherwise, access is prohibited, for example producing a fault or incrementing a counter or accumulator, such as a taint accumulator, which can be noted in an intrusion vector. For any aspect related to security, if a test is failed, the counter is incremented and placed in the intrusion vector.

An instruction can be specified in an instruction set which sets a capability. In various embodiments, the instruction can be implemented in software, hardware, the operating system, or the like. The instruction can operate in association with a capabilities vector. In some embodiments, the instruction can also or otherwise operate in association with a hint vector.

The capabilities vector can be associated with a pointer, an address, and an object. A highly basic capability is a lower bound and an upper bound. Other more complex capabilities can be implemented. In various implementations, the capabilities vector and the entitlement vector can be separate, or can be combined. Merging the capabilities vector and the entitlement vector enables software structuring.

The capabilities vector can be used to enable fine-grained permission. Fine-grained permission facilitates operations of multiple users or entities in a shared memory data base, enabling the multiple users to access storage such as disk

and to perform system calls, but limit access to data only to the user who owns the data or is authorized to access the data. Another benefit of fine-grained permissions is an ability to facilitate and improve security while multiplexing software threads onto hardware threads. In an example configuration, 64000 software threads are multiplexed onto only four hardware threads. Only a small portion of the software threads are running at one time with the remaining software threads idle. The software threads alternately run on the hardware threads, then go back to idle to allow other software threads to run.

A classic security hole in a database management is the inability to limit access to data for the different software threads multiplexed onto the hardware threads. A database typically does not allocate a hardware thread to a user. In typical database operation, a request is received and placed on a software thread so that users are multiplexed onto the software threads, an action giving very little protection. Better protection is attained by allocating each user to a separate process, a technique that is prohibitively expensive because the threads are expensive. Multiplexing the users onto software threads leaves a security hole because access to a particular user's data allowed while running the user's software thread on a hardware thread is not removed when the user's software thread is swapped out from the hardware thread. The access permission remains so access remains enabled. The depicted system solves the security hole by using capabilities.

In a non-capabilities system, any of the software threads can access the entire database at any time, including any data that has been placed in shared memory (unless a call out is made through the operating system to enable any of the threads to create I/O, a prohibitively expensive operation). Simple databases only have one peer thread so all threads can access any data. Many typical databases have 64 threads that can access any data in shared memory but only four threads that can access I/O. These systems sometimes have different privilege levels (for example, Intel's rings 0, 1, 2, 3) so specify compatibility. Most code runs in ring 3 and the kernel in ring 0. Rings 1 and 2 are generally not used although several databases have features that can run in ring 1 and ring 2 but are rare and used primarily for benchmarks (a benchmark hack).

In an example implementation that uses capabilities, generally a processor has 16 or 32 registers, some of which are addressing registers. A capability can be loaded to enable access to selected threads. A capability can be loaded to access a particular thread (owned by another user) into hardware thread 0, enabling running as that user. This is one type of context switch—to change the software thread that is executing on hardware thread 0. The capability registers can then be changed, a minor context switch and a change in privilege level. The action does not invalidate translation lookaside buffer (TLBs), but rather moves the permissions out of the TLB. The access control model is also changed. Capabilities can be used in this manner to change operations, guaranteeing only access to data and/or resources for which access is allowed by a permission-granting entity. Capabilities can guarantee a transitive exposure of only the data and/or resources of another user according to granted authorization. The technique is deterministic so that, by inspection, which accesses are possible is known.

Intrusion detection can use the concept of capabilities to implement fine-grained security.

Entitlements can be monitored using taint accumulation. Entitlements can be used to allocate resources. Entitlements can be defined as user-specified rights wherein a process is

31

entitled to a predetermined percentage of power or of time. A scheduler or chooser can monitor entitlement values and schedule the next highest priority process. A particular scheme can allocate modulo by bit to avoid starving a process with lower entitlement. In some conditions, the level of entitlement can be overridden or adjusted. Entitlement can be set according to a predetermined algorithm which defines a "fair share" for the processes, for example round-robin, history-based, randomized, and the like, which are efficient since a large history need not be accumulated. Thus, an efficient and inexpensive hardware implementation is possible. In some embodiments, a request for resources can be treated as a taint indicator and accumulated using a taint accumulator or taint vector to determine how to allocate among processes.

A metric can be specified which enables modification of a goal. A selected level of entitlement to resource consumption can be assigned to each process. One example scheme can be a short, low complexity method which is implemented while storing a limited operation history. For example, when running low on battery charge, a sequence 1-2-3-4-4-3-2-1 can be used to determine whether any of the processes is a resource glutton and can rank the processes on order of gluttony. The most gluttonous can be assigned the lowest priority. Another option can rank processes according to gluttony in combination with another factor of goodness (niceness). Processes can be ranked for the next cycle with the most gluttonous given last priority or can be ranked according to gluttony and one other nice system criterion. Monitoring and/or control can be performed highly efficiently if hardware, although either monitoring can be performed either in hardware or software in various embodiments. Power management units in CPUs can be used for monitoring, for example to monitor for increases or decreases in voltage or frequency, and for thread execution selection.

Capabilities can be used to perform monitoring and allocation of resources. For example, granting the capability to run video processing software can be combined with simultaneous granting of power capability.

Power is typically global to a process or to an individual CPU. Use of capabilities enables more refined control of power, for example power can be made specific to an object or library routine. With power global to a process, the process will continue to run in absence of a fault, a page fault, a disk access, or the like, and will run until blocked by the operating system scheduler, allowing high power consumption. Use of capabilities enables power to be controlled on a per-hardware thread granularity. Use of capabilities further enables power to be controlled specific to a per-hardware thread granularity for throttling power.

Processors can use instruction prefetch to improve execution speed by reducing wait states. The processor prefetches an instruction by request from main memory before the instruction is needed and, when retrieved from memory, placing the prefetched instruction in a cache. When needed, the instruction is quickly accessed from the cache. Prefetch can be used in combination with a branch prediction algorithm which anticipates results of execution to fetch predicted instructions in advance. Prefetches conventionally operate independently. In some embodiments, a processor disclosed herein can prefetch according to holistic monitoring of operating conditions such as voltage, frequency, and the like to more accurately determine or predict which instructions to prefetch.

The cache can be reconfigured dynamically, for example beginning with a single large, slow cache which can be

32

divided into a relatively small subcache and a larger sub-cache to enable faster operation. In embodiments disclosed herein, operating characteristics can be monitored to generate information for dynamic reconfiguring of the cache. In some embodiments, cache phenomena such as cache hits and misses can be handled as taint indicators for taint accumulation, for example using a taint vector, to facilitate handling of the cache. As a result of the monitored operating conditions, the cache can be selectively configured for slower or faster speed, larger and smaller cache subregions. In some conditions, part of the cache can be temporarily disabled, for example to save power. Monitoring of operating conditions can enable a suitable balance of considerations to determine whether part of the cache is to be disabled, for example determining whether the power saved in disabling part of the cache is appropriate in light of the power lost with a greater cache miss rate.

Disclosed system and method embodiments can use operating condition monitoring and holistic control at the level of calling an object. In an object-level paradigm, various objects or values (such as numbers, symbols, strings, and the like) can be combined to form other objects or values until the final result objects or values are obtained. New values can be formed from existing values by the application of various value-to-value functions, such as addition, concatenation, matrix inversion, and the like. Various objects have different impacts on system operations.

An example of an object which, when called, can have large consumption of power or other resources is video encoding which is a brute force, unintelligent algorithm which runs much more efficiently on dedicated hardware than a general CPU, and has real-time constraints. Video conferencing has similar real-time constraints.

Another object example is video games which perform many different tasks concurrently including processing geometry and processing video simultaneously, possibly processing speech for Skype communications, voice compression, input/output, and the like. Video games thus typically involve concurrent operation of multiple objects such as the game processing tasks and interface (Application Programming Interface, API) that perform different actions separately. The multiple objects are commonly run as separate threads, unless prohibitive due to the large amount of overhead in running threads that are not essential. Separate threads simplify programming.

In some configurations, applications, and conditions, multiple threads can be run wherein the threads need not be run in the same context.

Hyperthreading is a particular implementation of hardware threading. Software threading is a slightly different implementation of threading wherein the threads are often, but not always, related. In some implementations, a processor can include a GOAL register that can be used to set performance characteristics for particular threads. For example, if different routines (Skype, physics) are run in different threads, selected operating characteristics for the threads can be loaded into the GOAL register to give the threads separate issues. Allocating priority to the different threads can be difficult. In an illustrative system, priority to the threads can be allocated using a NICE utility which specifies acceptable performance for a particular operation and allows reduced priority in appropriate conditions for tasks that can be assigned lower priority with little or no consequence.

In an example implementation, priorities, particular types of priorities, and entitlements can be associated with particular library routines to facilitate management of relatively



heuristic phenomena. A library can be constituted wherein entitlements are assigned to individual library routines. The library includes information for adjusting the priority of threads, for example by identifying a phenomenon as a taint indication and accumulating taint indications. In some configurations or applications, the library can support hint vectors, such as branch prediction hints to specify whether static prediction should be taken or not taken. In some embodiments, the library can be configured to support NICE-type handling of a hint vector.

A process scheduler can be constituted to support prioritized entitlements and resource allocations upon calling selected libraries. A typical embodiment includes such support in software, although hardware support can also be implemented. For example, a network library can include library routines adapted for heavy network usage so that resources giving access to the network are more important processes to schedule. More entitlements are allocated to network-related resources. Libraries can also be configured to handle secondary priorities that change dynamically. For example, a sound card can have a greater power priority and have a pattern of operation wherein a process uses a network card and possibly other subsystems in combination with the sound card. Thus, the network card and other subsystems can also be allocated a higher priority. Similarly, for a process which performs less modeling and number computation in lieu of higher input/output operations and sending of information, a higher level of priority can be allocated to input/output resources.

Entitlements can be used to specify operations of a library. For example, a library with entitlement to run a predetermined number of floating point operations per second can, in response to a condition of executing instructions with few or no floating point computations, use the condition as a hint to power down floating point hardware, thus saving power. Thus, if computations include fixed point operations but no floating point operations, an a priori indicator can be generated designating that the floating point hardware is not needed in the near future and can be powered down. A process can call a library and, if known that a resource is not needed, the resource can be temporarily halted, thereby changing the entitlement level of that process with respect to the resource (for example a floating point unit) to a very low point.

In the illustrative example, the entitlement level of the process with respect to the floating point unit can be changed to very low because the resource is not needed for a foreseeable duration. The process thus indicates to other processes a willingness to relinquish access to the source, for example a willingness to be "nice" about allowing others to use the resource, so that access is deferred in favor of any other process that uses the resource, or the resource is shut down if not currently needed by another process.

Rather than have hardware determine demand for a resource after instructions have been executed, the illustrative system and method can use a call to a library or the result of making a call to the library as an indicator of entitlement niceness. This entitlement can be enforced in the manner of capabilities, for example by requesting access to a memory region, a request which may be denied. The library can give information regarding entitlement, thus giving a priori knowledge.

Resource allocation can also be managed using hints. An illustrative instruction that uses a hint is a hint that not much floating point computation is to be performed, a hint indicative of power demand. For example, hints to maintain power at a low level or to maintain power at a high level. An

exception can create problems when using hints, since a hint is not unwound in the event of an exception. For example, for a hint to maintain high power, an exception which changes the condition but does not reset the hint allows hardware to remain in a high power mode, potentially forever. Examples of problems with hint processing in conditions of context switching include problems with unlocking memory locations.

In contrast to entitlements, capabilities enable mechanisms to unwind.

Entitlement Vector can be used as part of or in affiliation with taint accumulation or a taint vector for managing resources. An entitlement vector can have multiple fields, for example including floating point, power, arithmetic logic unit (ALU), graphics triangle including any suitable entitlements, translation lookaside buffers TLBs, virtual memory usage, and the like. The entitlement vector can thus be used, for example, to power down the TLB as no longer relevant to operation, or to enable usage of a wide range of virtual memory.

Another field of the entitlement vector can specify scale. Examples of scale can be human scale, width of the accumulator, or any suitable scale. For example, for a finger print, a suitable scale can be no more than 2 MB.

A further field of the entitlement vector can be data path width, a similar concept to scale. A large instruction size, for example 1024 bits, wastes power, but typically only a portion of the bits are used at one time so that a desired subset of the bits can be activated, changing the data path width. The scale concept leads to the concept of a selected partial data path width. The data path width is part of the entitlement. For example, of 1024 bits logic can compute the number of bits actually needed and allocate that predetermined subset of bits, such as 128 bits. The data path field thus can be used to lower the data path width used of the available entitlement vector width, for example activating a super-accumulator data path width.

Referring to FIGS. 2A through 2Z, schematic flow diagrams depict an embodiment or embodiments of a method operable in a computing device adapted to manage security risk by monitoring taints and responding to predetermined taint conditions detected by the monitoring, for example by conditionally trapping based on the taint vector. An embodiment of a method **200** operable in a computing device for handling security risk, shown in FIG. 2A, can comprise providing **201** at least one taint vector; listing **202** in the at least one taint vector at least one of a plurality of taints indicative of potential security risk originating from at least one of a plurality of resources, and monitoring **203** at least one taint vector. The method **200** can further comprise responding **204** to a predetermined taint condition.

In some embodiments, as depicted in FIG. 2B, a method **205** can further comprise listing **206** in the at least one taint vector the plurality of taints comprising one or more of a plurality of distinct classes comprising a plurality of distinct sources, events, activities, and/or conditions.

In some embodiments, as shown in FIG. 2C, a method **207** can be implemented which further comprises selectively allocating **208** ones of a plurality of entries of the at least one taint vector to ones of the plurality of taints.

Referring to FIG. 2D, embodiments of a method **210** for handling security risk can further comprise trapping **211** based at least partly on a current value of an entry of the at least one taint vector.

In some embodiments, illustrated in FIG. 2E, the method **215** can further comprise arranging **216** at least one entry of the at least one taint vector using at least one accumulation

35

function, accumulating **217** the plurality of taints, and selecting **218** at least one accumulation function. For example, in various embodiments and/or in various conditions, accumulating **217** the plurality of taint indicators can be performed according to one or more selected functions of a plurality of accumulation functions. The accumulation functions can include comparing ones of the accumulated plurality of taint indicators to at least one predetermined threshold, performing power law analysis, performing a race function, performing a counting function, and the like. Suitable counting functions can include counting the number of taints, counting the number of taints per memory unit, counting the number of instructions tainted, counting the number of tainted instructions, counting the number of instructions written as a result of a taint, counting the number of data loads and stores, counting the number of memory accesses, counting the number of calls, counting the number of returns, counting the number of branches, counting the number of integer overflows, counting the number of network input/output events, counting the number of null pointer references, counting the number of buffer overruns/overflows, counting the number of repeated attempts to access a key, and the like.

In some embodiments and/or applications, the taints can be accumulated over time. For example, as depicted in FIG. **2F**, an embodiment of the method **220** can further comprise accumulating **221** taints in at least one entry of the at least one taint vector, and trapping **222** based at least partly on a current value of an accumulated entry of the at least one taint vector.

In various embodiments and/or applications, shown in FIG. **2G**, a method **223** can further comprise selecting **224** one or more of the plurality of taints for analysis. The taints can be selected, for example, from one or more circumstances or phenomena including a null pointer reference, an attempt to access a secured part of a processor, an attempt to access a secured resource, a buffer overrun, and an event originating in a region that raises suspicion. Other example taints can include a fault, an integer overflow, a plurality of taint indicators that exceeds at least one predetermined threshold, a taint indicated by power law analysis, a taint indicated by a race function, and an attempt to access a key, and a variety of other circumstances or phenomena.

Further embodiments, for example as illustrated in FIG. **2H**, a method **225** can further comprise accumulating **226** taints in at least one entry of the at least one taint vector, comparing **227** a current value of an accumulated entry of the at least one taint vector to a threshold, and trapping **228** based at least partly on the comparison.

In some embodiments a target function can be applied in monitoring and analyzing taints. For example, as shown in FIG. **2I**, a method **230** can further comprise applying **231** at least one function to the at least one entry of the at least one taint vector.

In particular embodiments, as shown in FIG. **2J**, a method **232** can further comprise selecting **233** the at least one function from a group consisting of weights, masks, sums, combinations, arithmetic functions, logical operations, and transforms.

As shown in FIG. **2K**, a method **235** can perform taint vector-based trapping for intrusion detection. An illustrative method **235** can further comprise setting **236** a threshold for application to at least one entry of the at least one taint vector, comparing **237** the one or more entries of the at least one taint vector to the threshold, and trapping **238** based at least partly on the comparison.

36

Similarly, referring to FIG. **2L**, a method **240** can further comprise setting **241** a threshold for application to at least one entry of the at least one taint vector mutually distinctively for a plurality of affiliates, system characteristics, sources, events, activities, and/or conditions.

Similarly, referring to FIG. **2M**, a method **242** of handling security can further comprise setting **243** a threshold for application to at least one entry of the at least one taint vector specifically to the at least one entry.

Thresholds can vary by affiliate, current system characteristics or activities, and the like. Accordingly, as depicted in FIG. **2N**, an embodiment of a method **245** for handling security can further comprise setting **246** a threshold for application to at least one entry of the at least one taint vector uniformly for application to similar affiliates.

Taints for memory may be located in different places, organized in various manners among different memory locations or types, and/or associated with different amounts of memory. Hence, referring to FIG. **2O**, a method **247** can further comprise setting **248** a threshold for application to at least one entry of the at least one taint vector universally wherein all entries are compared to the threshold for a plurality of affiliates, system characteristics, sources, events, activities, and/or conditions.

Furthermore, as shown in FIG. **2P**, a method **250** can further comprise setting **251** a threshold for application to a sum of entries of the at least one taint vector.

In various embodiments, thresholds can be dynamically modified based on operations and received taints. As illustrated in FIG. **2Q**, a method **252** can further comprise modifying **253** a threshold based on variation of tolerance level by application of a predetermine weight function, and applying **254** the modified threshold to at least one entry of the at least one taint vector.

Similarly, as depicted in FIG. **2R**, a method **255** can further comprise modifying **256** a threshold based on a tolerance level selected on the basis of consequences of a security risk event, and applying **257** the modified threshold to at least one entry of the at least one taint vector.

Referring to FIG. **2S**, a method **260** of handling security risk can further comprise comparing **261** at least one entry of the at least one taint vector to a plurality of thresholds, and responding **262** to the comparison.

A taint indication can originate from outside of a system being monitored or from internal to the system. A taint can originate from local or remote software. A taint may arise from suspicious operations or faults or from hints. A taint can be determined, for example, by an attempt to write to a web browser. Accordingly, as depicted in FIG. **2T**, a method **265** can further comprise monitoring **266** taints, creating **267** a trust profile based on the monitoring, and trapping **268** to a software process based at least partly on determination of a suspicious condition.

Similarly, as shown in FIG. **2U**, the method **270** can further comprise monitoring **271** taints in network input/output operations, determining **272** a network input/output condition of an attempt of malware to communicate to a malware operator, and trapping **273** to a software process based at least partly on the determination.

Likewise, as illustrated in FIG. **2V**, a method **275** can further comprise monitoring **276** taints using a hardware device, and inserting **277** initial taint notifications using a software process.

Referring to FIG. **2W**, a method **280** of handling security risk can further comprise updating **281** the taint vector, processing **282** the taint vector, determining **283** a security

37

risk condition based on the processing of the taint vector, and trapping **284** in response to the security risk condition.

Taints and responses to taints can be dynamically adjusted through operations over time. Accordingly, as illustrated in FIG. 2X, a method **285** of handling security risk can further comprise acquiring **286** a history of the ones of the at least one taint vector in a feedback loop, monitoring **287** the history of the ones of the at least one taint vector in the feedback loop, and correlating **288** taints with responses to the taints.

Similarly, as shown in FIG. 2Y, a method **290** of handling security risk can further comprise responding **291** to at least one security risk event with at least one response, and selecting **292** at least one response. Responses can include ignoring one or more security risk events, logging the security risk event(s), displaying a notification, or displaying a warning message. Other example responses can include generating an alarm, preventing a memory and/or register write, modifying operating frequency, modifying operating voltage, or modifying an operating parameter. Still other responses can include performing a system call, calling a trap and/or exception, terminating operation of selected resources, activating a system shutdown, and others.

Referring to FIG. 2Z, a method **295** of handling security risk can further comprise configuring **296** one or more of the at least one taint vector as a composite taint vector that correlates a taint source and a taint activity type.

Referring to FIGS. 3A and 3B, embodiments comprise a computer program product **300** adapted to manage security risk by accumulating and monitoring taint indications. The computer program product includes a computer-readable storage medium **306** bearing program instructions. The program instructions are operable to perform a process in a computing device. The computer program product can be constituted as any combination of one or more computer usable or computer readable medium(s), for example but not limited to, communication, electronic, semiconductor, magnetic, optical, electromagnetic, infrared, in the form of propagation medium, system, apparatus, device, or the like. Specific examples of the computer-readable medium may include, are not limited to, a wired connection, a wireless connection, Internet or an intranet transmission media, an optical fiber, a magnetic storage device, a portable diskette, a hard disk, a portable compact disc read-only memory (CDROM), an optical storage device, a random access memory (RAM), a read-only memory (ROM), an erasable programmable read-only memory (EPROM), Flash memory, or the like. Similarly, the computer-usable or computer-readable medium can be a visual display such as paper, poster, screen view, that can be visually or electronically captured such as by optical scanning of a medium, then compiled, interpreted, or otherwise processed.

As shown in FIG. 3A, the computer program product **300** can comprise program instructions **310** operable to provide at least one taint vector, program instructions **312** operable to list in the at least one taint vector at least one of a plurality of taints indicative of potential security risk originating from at least one of a plurality of resources, and program instructions **314** operable to monitor the at least one taint vector. The computer program product **300** can further comprise program instructions **316** operable to respond to a predetermined taint condition.

In some embodiments, the computer program product **300** can further comprise program instructions **318** operable to list in the at least one taint vector the plurality of taints comprising one or more of a plurality of distinct classes comprising a plurality of distinct sources, events, activities,

38

and/or conditions; and program instructions **320** operable to selectively allocate ones of a plurality of entries of the at least one taint vector to ones of the plurality of taints. The computer program product **300** can further comprise program instructions **322** operable to trap based at least partly on a current value of an entry of the at least one taint vector.

In various embodiments and/or applications, the computer program product **300** can further comprise program instructions **324** operable to accumulate taints in at least one entry of the at least one taint vector, and program instructions **326** operable to compare a current value of an accumulated entry of the at least one taint vector to a threshold. The computer program product **300** can further comprise program instructions **328** operable to trap based at least partly on a current value of an accumulated entry of the at least one taint vector and/or based at least partly on the comparison.

As shown in FIG. 3B, the computer program product **300** can comprise program instructions **330** operable to select the at least one function from a group consisting of weights, masks, sums, combinations, arithmetic functions, logical operations, and transforms; and program instructions **332** operable to apply at least one function to the at least one entry of the at least one taint vector.

In some embodiments, the computer program product **300** can further comprise program instructions **334** operable to set a threshold for application to at least one entry of the at least one taint vector for selected ones of a plurality of affiliates, system characteristics, sources, events, activities, and/or conditions. The threshold can be set selectively from among mutually distinctively, specifically to the at least one entry of the at least one taint vector, uniformly for application to similar affiliates, and universally wherein all entries are compared to the threshold for a plurality of affiliates, system characteristics, sources, events, activities, and/or conditions. The computer program product **300** can further comprise program instructions **336** operable to compare the at least one entry of the at least one taint vector to the threshold, and program instructions **338** operable to trap based at least partly on the comparison.

In various embodiments and/or applications, the computer program product **300** can further comprise program instructions **340** operable to modify a threshold based on variation of tolerance level on the basis of consequences of a security risk event or by application of a predetermine weight function, and program instructions **342** operable to apply the modified threshold to at least one entry of the at least one taint vector.

Referring to FIGS. 4A and 4B, schematic block diagrams illustrate embodiments of a computing system **400** adapted to manage security risk by accumulating and monitoring taint indications, and, for some embodiments, facilitating response to predetermined taint conditions that are detected during the monitoring. An embodiment of the computing system **400** can comprise means **430** for providing at least one taint vector **404**, means **432** for listing in the at least one taint vector **404** at least one of a plurality of taints **405** indicative of potential security risk originating from at least one of a plurality of resources **402**, and means **434** for monitoring the at least one taint vector **404**. The computing system **400** can further comprise means **436** for responding to a predetermined taint condition.

In some embodiments, the computing system **400** can further comprise means **438** for listing in the at least one taint vector **404** the plurality of taints **105** comprising one or more of a plurality of distinct classes **410** comprising a plurality of distinct sources **412**, events **414**, activities **416**, and/or conditions **418**. The computing system **400** can

39

further comprise means **440** for selectively allocating one more of a plurality of entries **420** of the at least one taint vector **404** to ones of the plurality of taints **405**, and means **442** for trapping based at least partly on a current value of an entry **420** of the at least one taint vector **404**.

In various embodiments and/or conditions, security risk can be assessed via one or more actions selected from actions such as determining whether any elements are greater than a predetermined threshold, determining whether all elements are greater than a predetermined threshold, determining whether the sum of some elements is greater than a predetermined threshold, determining whether the sum of all elements is greater than a predetermined threshold, and the other similar suitable actions. Accordingly, the computing system **400** can further comprise means **444** for accumulating taints **405** in at least one entry **420** of the at least one taint vector **404**, and means **446** for comparing a current value of an accumulated entry **420** of the at least one taint vector **404** to a threshold. The computing system **400** can further comprise means **448** for trapping based at least partly on a current value of an accumulated entry **420** of the at least one taint vector **404** and/or based at least partly on the comparison.

The computing system **400** enables usage of various target functions for analysis of taints **405** to optimize monitoring and detection of security risks. Accordingly, the computing system **400** can further comprise means **450** for selecting the at least one function **422** from a group consisting of weights, masks, sums, combinations, arithmetic functions, logical operations, and transforms. The computing system **400** can further comprise means **452** for applying at least one function **422** to the at least one entry **420** of the at least one taint vector **404**.

Referring to FIG. 4B, the computing system **400** can be configured to dynamically adapt thresholds to operations during execution. Accordingly, the computing system **400** can further comprise means **454** for setting a threshold for application to at least one entry of the at least one taint vector **404** for selected affiliates, system characteristics, sources **412**, events **414**, activities **416**, and/or conditions **418**. The threshold can be set selectively from among mutually distinctively, specifically to the at least one entry **420** of the at least one taint vector **404**, uniformly for application to similar affiliates, and universally wherein all entries **420** are compared to the threshold for a plurality of affiliates, system characteristics, sources **412**, events **414**, activities **416**, and/or conditions **418**. The computing system **400** can further comprise means **456** for comparing the at least one entry **420** of the at least one taint vector **404** to the threshold, and means **458** for trapping based at least partly on the comparison.

Similarly, the computing system can further comprise means **460** for modifying a threshold based on variation of tolerance level on the basis of consequences of a security risk event or by application of a predetermined weight function, and means **462** for applying the modified threshold to at least one entry **420** of the at least one taint vector **404**.

Taints can be allocated to a taint vector in any suitable manner. For example, taints can be allocated at a selected granularity selected from allocations including allocating taints by memory page, allocating taints by byte, allocating taints by word, allocating taints by memory block, allocating taints by hardware process identifier (PID), and allocating taints to enable a cross-thread taint. Additional allocations can include allocating taints among hardware devices, allocating taints by component, allocating taints by software component, and the like.

40

In various embodiments and/or conditions, one or more decay options can be selected from a plurality of decay options such as applying decay after a predetermined number of operations to avoid triggering on outlying events, setting decay to account for rare and spurious events with a probability of occurrence by chance during long term monitoring, incrementing/decrementing using a single vector, and subtracting a predetermined number. Additional decay options can include shifting a taint vector in an interval of time, shifting a taint vector at a predetermined instruction count, shifting a taint vector at a predetermined processor cycle count, copying a taint vector periodically to memory to maintain an old version while incrementing/decrementing to enable restoration following an invalid or error condition, imposing decay that balances accumulation, applying decay periodically, applying decay with a varying period that varies based on a sensitivity meter, applying decay with a varying period that varies based on environment, applying decay with a varying period that varies based on activity type, applying decay according to a programmable parameter at a programmable rate, and the like.

In various embodiments and/or conditions, responding to security risk in response to detection of the at least one security risk event can be one or more responses selected from a group of responses that range from relatively minor informational actions to actions which can moderately or substantially change system operations, or even terminate some or all system operations. Minor or informational responses can include including ignoring the at least one security risk event, logging the at least one security risk event, displaying a notification, displaying a warning message, generating an alarm, and the like. Responses affecting system operations can include preventing a memory and/or register write, modifying operating frequency, modifying operating voltage, modifying another operating parameter, performing a system call, and others. More drastic responses that can moderately or substantially affect operations can include calling a trap and/or exception, terminating operation of selected resources, activating a system shutdown, and the like.

Terms “substantially”, “essentially”, or “approximately”, that may be used herein, relate to an industry-accepted variability to the corresponding term. Such an industry-accepted variability ranges from less than one percent to twenty percent and corresponds to, but is not limited to, materials, shapes, sizes, functionality, values, process variations, and the like. The term “coupled”, as may be used herein, includes direct coupling and indirect coupling via another component or element where, for indirect coupling, the intervening component or element does not modify the operation. Inferred coupling, for example where one element is coupled to another element by inference, includes direct and indirect coupling between two elements in the same manner as “coupled”.

The illustrative pictorial diagrams depict structures and process actions in a manufacturing process. Although the particular examples illustrate specific structures and process acts, many alternative implementations are possible and commonly made by simple design choice. Manufacturing actions may be executed in different order from the specific description herein, based on considerations of function, purpose, conformance to standard, legacy structure, and the like.

While the present disclosure describes various embodiments, these embodiments are to be understood as illustrative and do not limit the claim scope. Many variations, modifications, additions and improvements of the described

41

embodiments are possible. For example, those having ordinary skill in the art will readily implement the steps necessary to provide the structures and methods disclosed herein, and will understand that the process parameters, materials, shapes, and dimensions are given by way of example only. 5 The parameters, materials, and dimensions can be varied to achieve the desired structure as well as modifications, which are within the scope of the claims. Variations and modifications of the embodiments disclosed herein may also be made while remaining within the scope of the following 10 claims.

What is claimed is:

1. A computing system comprising:

one or more input interfaces configured to receive information from a plurality of computing resources; 15

one or more processors configured to monitor one or more taints associated with the information received from at least one of the plurality of computing resources, the one or more processors including at least:

at least one taint vector including a plurality of vector fields operated upon by one or more instructions in parallel to monitor and respond to at least one of a plurality of taints indicative of potential security risk originating from at least one of the plurality of computing resources, the at least one of the plurality of taints independently associated with the at least one of the plurality of computing resources, wherein the plurality of vector fields of one or more of the at least one taint vector include at least: 20

at least one vector field identifying at least one of the plurality of computing resources; and 30

at least one vector field corresponding to at least one of sources, events, conditions, or suspicious activities associated with the identified at least one of the plurality of computing resources; 35

at least one taint bias vector; and

response logic operable to monitor the at least one taint vector and respond to a predetermined taint condition in a predetermined manner, wherein the response logic is configured to respond to one or more predetermined instructions in parallel by recursively adding the at least one taint bias vector to the at least one taint vector; 40

wherein the predetermined manner includes one or more of ignoring a security risk event, logging the at least one security risk event, displaying a notification, displaying a warning message, generating an alarm, raising an exception, preventing writing by or to one or more of the plurality of computing resources, trapping one or more operations attempted by one or more of the plurality of computing resources, modifying operating frequency, modifying operating voltage, modifying an operating parameter, performing a system call, terminating a particular process, or ending one or more operations of one or more of the plurality of computing resources. 55

2. The computing system according to claim 1 wherein: the plurality of taints include one or more of a plurality of distinct classes corresponding to a plurality of distinct sources, events, activities, or conditions.

3. The computing system according to claim 1 wherein: one or more of the at least one taint vector include one or more entries selectively allocated to one or more of the plurality of taints. 60

4. The computing system according to claim 1 wherein: the response logic is configured to trap based at least partly on a current value of an entry of the at least one taint vector. 65

42

5. The computing system according to claim 1 wherein: the response logic is configured to accumulate taints in at least one entry of the at least one taint vector to trap based at least partly on a current value of an accumulated entry of the at least one taint vector.

6. The computing system according to claim 1 wherein: the response logic is configured accumulate taints in at least one entry of the at least one taint vector to trap based at least partly on a comparison of a current value of an accumulated entry of the at least one taint vector to a threshold.

7. The computing system according to claim 1 wherein: the response logic is configured apply at least one function to the at least one entry of the at least one taint vector.

8. The computing system according to claim 7 wherein: the at least one function is selected from one or more of weights, masks, sums, combinations, arithmetic functions, logical operations, or transforms.

9. The computing system according to claim 1 further comprising:

thresholding logic operationally coupled to the response logic and configured to set a threshold for application to at least one entry of the at least one taint vector, wherein:

the response logic is configured to trap based at least partly on a comparison of the at least one entry of the at least one taint vector to the threshold.

10. The computing system according to claim 9 wherein: the thresholding logic is configured to set a threshold for application to at least one entry of the at least one taint vector mutually distinctively for a plurality of affiliates, system characteristics, sources, events, activities, or conditions.

11. The computing system according to claim 9 wherein: the thresholding logic is configured to set a threshold for application to at least one entry of the at least one taint vector specifically to the at least one entry.

12. The computing system according to claim 9 wherein: the thresholding logic is configured to set a threshold for application to at least one entry of the at least one taint vector uniformly for application to similar affiliates.

13. The computing system according to claim 9 wherein: the thresholding logic is configured to set a threshold for application to at least one entry of the at least one taint vector universally wherein all entries are compared to the threshold for a plurality of affiliates, system characteristics, sources, events, activities, or conditions.

14. The computing system according to claim 9 wherein: the thresholding logic is configured to set a threshold for application to a sum of entries of the at least one taint vector.

15. The computing system according to claim 9 wherein: the thresholding logic is configured to set or modify a threshold for application to at least one entry of the at least one taint vector based on variation of tolerance level by application of a predetermined weight function.

16. The computing system according to claim 9 wherein: the thresholding logic is configured to set or modify a threshold for application to at least one entry of the at least one taint vector based on a tolerance level selected on the basis of consequences of a security risk event.

17. The computing system according to claim 9 wherein: the response logic is configured to respond to at least one comparison between the at least one entry of the at least one taint vector and a plurality of thresholds.

## 43

18. The computing system according to claim 1 further comprising:  
 monitoring logic operationally coupled to the response logic and configured to monitor taints and create a trust profile based on the monitoring, wherein:  
 the response logic is configured to trap to a software process based at least partly on determination of a suspicious condition indicated at least in part using the trust profile.
19. The computing system according to claim 1 further comprising:  
 monitoring logic operationally coupled to the response logic and configured to monitor taints in network input/output operations, wherein:  
 the response logic is configured to trap to a software process based at least partly on determination of a network input/output condition of an attempt of malware to communicate to a malware operator.
20. The computing system according to claim 1 further comprising:  
 monitoring logic operationally coupled to the response logic and configured to monitor taints using a hardware device configured to monitor the taints autonomously of software; wherein:  
 the response logic is configured to insert initial taint notifications using a software process.
21. The computing system according to claim 1 further comprising:  
 monitoring logic operationally coupled to the response logic and configured to update the taint vector, process the taint vector, and determine a security risk condition based on the processing of the taint vector; wherein:  
 the response logic is configured to trap in response to the security risk condition.
22. The computing system according to claim 1 further comprising:  
 monitoring logic operationally coupled to the response logic and configured to acquire and monitor a history of the one or more of the at least one taint vector in a feedback loop that correlates taints with responses to the taints.
23. The computing system according to claim 1 further comprising:  
 monitoring logic operationally coupled to the response logic and configured to accumulate the plurality of taints arranged as at least one entry of the at least one taint vector using at least one accumulation function selected from:  
 comparing one or more of the accumulated plurality of taints to at least one predetermined threshold;  
 performing power law analysis;  
 performing a race function;  
 counting a number of taints;  
 counting a number of taints per memory unit;  
 counting a number of instructions tainted;  
 counting a number of tainted instructions;  
 counting a number of instructions written as a result of one or more taints;  
 counting a number of data loads and stores;  
 counting a number of memory accesses;  
 counting a number of calls;  
 counting a number of returns;  
 counting a number of branches;  
 counting a number of integer overflows;  
 counting a number of network input/output events;  
 counting a number of null pointer references;  
 counting a number of buffer overruns/overflows; or  
 counting a number of repeated attempts to access a key.

## 44

24. The computing system according to claim 1 wherein:  
 the response logic is configured to respond to at least one security risk event with at least one response selected from:  
 ignoring the at least one security risk event;  
 logging the at least one security risk event;  
 displaying a notification;  
 displaying a warning message;  
 generating an alarm;  
 preventing a memory or register write;  
 modifying operating frequency;  
 modifying operating voltage;  
 modifying an operating parameter;  
 performing a system call;  
 calling a trap or exception;  
 terminating operation of selected resources; or  
 activating a system shutdown.
25. The computing system according to claim 1 wherein one or more of the plurality of taints are selected from:  
 a null pointer reference;  
 an attempt to access a secured part of a processor;  
 an attempt to access a secured resource;  
 a buffer overrun;  
 an event originating in a region that raises suspicion;  
 a fault;  
 an integer overflow;  
 a plurality of taint indicators that exceeds at least one predetermined threshold;  
 a taint indicated by power law analysis;  
 a taint indicated by a race function; or  
 an attempt to access a key.
26. The computing system according to claim 1 wherein:  
 one or more of the at least one taint vector include a composite taint vector that correlates a taint source and a taint activity type.
27. The computing system according to claim 1 further comprising:  
 a composite taint vector including at least two taint vectors each including a plurality of bits corresponding to at least one of one or more sources, one or more events, one or more activities, one or more conditions, or one or more suspicious activities in association between the at least two taint vectors; wherein  
 the at least two taint vectors each correspond to a data source; and  
 the composite taint vector is configured for monitoring and tracking the associated at least one of one or more sources, one or more events, one or more activities, one or more conditions, or one or more suspicious activities corresponding to the at least two taint vectors.
28. The computing system according to claim 1 further comprising:  
 a composite taint vector including at least two taint vectors each including a plurality of bits corresponding identically to one or more of a plurality of distinct classes for the at least two taint vectors, the at least two taint vectors including at least one taint vector that is decayed over a selected number of operations and at least one taint vector that is maintained for restoration in reaction to an invalid or error condition.
29. The computing system according to claim 1 further comprising:  
 at least one timer register configured to change at a predetermined rate; wherein the at least one taint vector is configured to update in a predetermined manner in response to receipt of the one or more of the plurality

45

of taint indicators and in response to a predetermined condition of the at least one timer register.

30. The computing system according to claim 1 wherein: the at least one taint vector is configured into a plurality of portions that update independently according to one or more distinct accumulation functions in response to receipt of the one or more of the plurality of taint indicators corresponding selectively to one or more distinct taint conditions and one or more sources.

31. The computing system according to claim 1 further comprising:

hardware threading circuitry configured for at least one of simultaneous multithreading (SMT) or hyperthreading; wherein the at least one taint vector is configured into a plurality of portions that update independently according to one or more distinct accumulation functions in response to receipt of the one or more of the plurality of taint indicators corresponding selectively to one or more distinct threads executing on the hardware threading circuitry.

32. The computing system according to claim 1 wherein the at least one taint vector including a plurality of vector fields operated upon by one or more instructions in parallel to monitor and respond to at least one of a plurality of taints indicative of potential security risk originating from at least one of a plurality of computing resources includes at least:

at least one vector field identifying at least one of the plurality of computing resources; and

at least one vector field corresponding to at least one of sources, events, conditions, or suspicious activities associated with the identified at least one of the plurality of computing resources,

the at least one taint vector configured to set a hierarchy of suspicion based at least partially on the at least one of sources, events, conditions, or suspicious activities.

33. The computing system according to claim 1 wherein one or more of the at least one taint vector including a plurality of vector fields operated upon by one or more instructions in parallel to monitor and respond to at least one of a plurality of taints indicative of potential security risk originating from at least one of a plurality of computing resources includes at least:

at least one vector field identifying at least one of the plurality of computing resources; and

at least one vector field corresponding to at least one of sources, events, conditions, or suspicious activities associated with the identified at least one of the plurality of computing resources, wherein the at least one taint vector is configured to respond to one or more predetermined instructions in parallel by applying a hint associated with the at least one of sources, events, conditions, or suspicious activities.

34. The computing system according to claim 1 wherein the response logic operable to monitor the at least one taint vector and respond to a predetermined taint condition comprises:

response logic operable to determine the at least one taint bias vector to selectively increase or decrease a level of sensitivity to security risk based at least in part on the at least one of sources, events, conditions, or suspicious activities associated with the identified at least one of the plurality of computing resources.

35. The computing system according to claim 1, wherein the computing resources include:

at least one of a network, a system, a processor, memory, a register, hardware, microarchitecture, floating point circuitry, input/output circuitry, video circuitry, audio

46

circuitry, a software system, software, an operating system, a library, a library call, a library function, a software object, a virtual entity, bandwidth, or power.

36. A method operable in a computing device configured at least partially in hardware for handling security risk comprising:

receiving information from a plurality of computing resources;

monitoring one or more taints associated with the information received from at least one the plurality of computing resources;

providing at least one taint vector including a plurality of vector fields operated upon by one or more instructions;

executing the one or more instructions in parallel to monitor and respond to the at least one taint vector including a plurality of vector fields associated with at least one of a plurality of taints indicative of potential security risk originating from at least one of the plurality of computing resources, wherein the plurality of vector fields of one or more of the at least one taint vector including a plurality of vector fields include at least:

at least one vector field identifying at least one of the plurality of computing resources; and

at least one vector field corresponding to at least one of sources, events, conditions, or suspicious activities associated with the identified at least one of the plurality of computing resources;

independently associating the at least one of the plurality of taints with the at least one of the plurality of computing resources;

monitoring the at least one taint vector; and

responding to a predetermined taint condition including at least responding to one or more predetermined instructions in a predetermined manner in parallel by recursively adding at least one taint bias vector to the at least one taint vector;

wherein the predetermined manner includes one or more of ignoring a security risk event, logging the at least one security risk event, displaying a notification, displaying a warning message, generating an alarm, raising an exception, preventing writing by or to one or more of the plurality of computing resources, trapping one or more operations attempted by one or more of the plurality of computing resources, modifying operating frequency, modifying operating voltage, modifying an operating parameter, performing a system call, terminating a particular process, or ending one or more operations of one or more of the plurality of computing resources.

37. The method according to claim 36 further comprising: listing in the at least one taint vector the plurality of taints including one or more of a plurality of distinct classes including one or more of a plurality of distinct sources, events, activities, or conditions.

38. The method according to claim 36 further comprising: selectively allocating one or more of a plurality of entries of the at least one taint vector to one or more of the plurality of taints.

39. The method according to claim 36 further comprising: trapping based at least partly on a current value of an entry of the at least one taint vector.

40. The method according to claim 36 further comprising: accumulating taints in at least one entry of the at least one taint vector; and

trapping based at least partly on a current value of an accumulated entry of the at least one taint vector.

47

41. The method according to claim 36 further comprising:  
 accumulating taints in at least one entry of the at least one  
 taint vector;  
 comparing a current value of an accumulated entry of the  
 at least one taint vector to a threshold; and  
 trapping based at least partly on the comparison. 5
42. The method according to claim 36 further comprising:  
 applying at least one function to the at least one entry of  
 the at least one taint vector.
43. The method according to claim 42 further comprising: 10  
 selecting the at least one function from weights, masks,  
 sums, combinations, arithmetic functions, logical  
 operations, or transforms.
44. The method according to claim 36 further comprising:  
 setting a threshold for application to at least one entry of 15  
 the at least one taint vector;  
 comparing the at least one entry of the at least one taint  
 vector to the threshold; and  
 trapping based at least partly on the comparison.
45. The method according to claim 36, wherein the 20  
 computing resources include:  
 at least one of a network, a system, a processor, memory,  
 a register, hardware, microarchitecture, floating point  
 circuitry, input/output circuitry, video circuitry, audio  
 circuitry, a software system, software, an operating 25  
 system, a library, a library call, a library function, a  
 software object, a virtual entity, bandwidth, or power.
46. A computing system comprising:  
 one or more input interfaces configured to receive infor- 30  
 mation from a plurality of computing resources;  
 one or more processors configured to monitor one or more  
 taints associated with the information received from at  
 least one of the plurality of computing resources, the  
 one or more processors including at least:  
 at least one taint vector including a plurality of vector 35  
 fields operated upon by one or more instructions;  
 means for executing the one or more instructions in  
 parallel to monitor and respond to the at least one taint  
 vector including a plurality of vector fields associated  
 with at least one of a plurality of taints indicative of 40  
 potential security risk originating from at least one of

48

- the plurality of computing resources, wherein the plu-  
 rality of vector fields of one or more of the at least one  
 taint vector include at least:  
 at least one vector field identifying at least one of the  
 plurality of computing resources; and  
 at least one vector field corresponding to at least one of  
 sources, events, conditions, or suspicious activities  
 associated with the identified at least one of the  
 plurality of computing resources;  
 means for independently associating the at least one of  
 the plurality of taints with the at least one of the  
 plurality of computing resources;  
 means for monitoring the at least one taint vector; and  
 means for responding to a predetermined taint condition  
 in a predetermined manner including at least respond-  
 ing to one or more predetermined instructions in par-  
 allel by recursively adding at least one taint bias vector  
 to the at least one taint vector;  
 wherein the predetermined manner includes one or  
 more of ignoring a security risk event, logging the at  
 least one security risk event, displaying a notifica-  
 tion, displaying a warning message, generating an  
 alarm, raising an exception, preventing writing by or  
 to one or more of the plurality of computing  
 resources, trapping one or more operations attempted  
 by one or more of the plurality of computing  
 resources, modifying operating frequency, modify-  
 ing operating voltage, modifying an operating  
 parameter, performing a system call, terminating a  
 particular process, or ending one or more operations  
 of one or more of the plurality of computing  
 resources.
47. The computing system according to claim 46, wherein  
 the computing resources include:  
 at least one of a network, a system, a processor, memory,  
 a register, hardware, microarchitecture, floating point  
 circuitry, input/output circuitry, video circuitry, audio  
 circuitry, a software system, software, an operating  
 system, a library, a library call, a library function, a  
 software object, a virtual entity, bandwidth, or power.

\* \* \* \* \*